Competency Based Learning Material (CBLM)

AI in Immersive Technology

Level-6

Module: Implement of Deep Learning

Code: OU-ICT-AIIT-04-L6-V1

National Skills Development Authority
Chief Advisor's Office
Government of the People's Republic of Bangladesh

Copyright

National Skills Development Authority

Chief Advisor's Office

Level: 10-11, Biniyog Bhaban,

E-6 / B, Agargaon, Sher-E-Bangla Nagar Dhaka-1207, Bangladesh.

Email: ec@nsda.gov.bd

Website: www.nsda.gov.bd.

National Skills Portal: http:\\skillsportal.gov.bd

This Competency Based Learning Materials (CBLM) on "Implement of Deep Learning" under the AI in Immersive Technology, Level-6" qualification is developed based on the national competency standard approved by National Skills Development Authority (NSDA)

This document is to be used as a key reference point by the competency-based learning materials developers, teachers/trainers/assessors as a base on which to build instructional activities.

It serves as the document for providing training consistent with the requirements of industry in order to meet the qualification of individuals who graduated through the established standard via competency-based assessment for a relevant job.
This document has been developed by NSDA in association with industry representatives, academia, related specialist, trainer and related employee.

Public and private institutions may use the information contained in this CBLM for activities benefitting Bangladesh.

Approved by the Authority….. meeting held on **.....**

How to use this Competency Based Learning Material (CBLM)

The module, Implement of Deep Learning contains training materials and activities for you to complete. These activities may be completed as part of structured classroom activities or you may be required you to work at your own pace. These activities will ask you to complete associated learning and practice activities in order to gain knowledge and skills you need to achieve the learning outcomes.

1. Review the **Learning Activity** page to understand the sequence of learning activities you will undergo. This page will serve as your road map towards the achievement of competence.
2. Read the **Information Sheets.** This will give you an understanding of the jobs or tasks you are going to learn how to do. Once you have finished reading the **Information Sheets** complete the questions in the **Self-Check.**
3. **Self-**Checks are found after each **Information Sheet**. **Self-Checks** are designed to help you know how you are progressing. If you are unable to answer the questions in the **Self-Check** you will need to re-read the relevant **Information Sheet**. Once you have completed all the questions check your answers by reading the relevant **Answer Keys** found at the end of this module.
4. Next move on to the **Job Sheets. Job Sheets** provide detailed information about *how to do the job* you are being trained in. Some **Job Sheets** will also have a series of **Activity Sheets**. These sheets have been designed to introduce you to the job step by step. This is where you will apply the new knowledge you gained by reading the Information Sheets. This is your opportunity to practise the job. You may need to practise the job or activity several times before you become competent.
5. Specification **sheets**, specifying the details of the job to be performed will be provided where appropriate.

A review of competency is provided on the last page to help remind if all the required assessment criteria have been met. This record is for your own information and guidance and is not an official record of competency

When working though this Module always be aware of your safety and the safety of others in the training room. Should you require assistance or clarification please consult your trainer or facilitator.

When you have satisfactorily completed all the Jobs and/or Activities outlined in this module, an assessment event will be scheduled to assess if you have achieved competency in the specified learning outcomes. You will then be ready to move onto the next Unit of Competency or Module

Table of Contents

Module Content

Unit of Competency	**Implement of Deep Learning**
Unit Code	OU-ICT-AIIT-04-L6-V1
Module Title	**Exercise Deep Learning (DL)**
Module Descriptor	This unit covers the knowledge, skills and attitude required to apply Math Skills for Machine. It specifically includes the requirements of applying statistical measures, using multivariable calculus, applying Linear Algebra, and using optimization methods.
Nominal Hours	60 Hours
Learning Outcome	After completing the practice of the module, the trainees will be able to perform the following jobs: 1. Exercise Deep Learning (DL) 2. Analyze features 3. Implement DL features 4. Perform project development activities

Assessment Criteria

1. Fundamentals of DL is interpreted
2. Activities for DL Tools and libraries are performed
3. Data preparation is implemented
4. Manual and automatic data labeling are interpreted
5. Automatic labeling techniques are implemented
6. Feature extraction is implemented
7. Visualization of word vectors are implemented with word cloud, histogram, heatmap, plots and tableau
8. Embedding models are exercised
9. Pre-trained word embedding is implemented
10. ANN and CNN are comprehended
11. CNN Variations are implemented
12. Optimization of hyperparameters is implemented
13. Recurrent neural networks are implemented
14. Ensemble of DL Models is implemented
15. Projects are developed using DL

16. Transformer based models are interpreted
17. Specific models are designed
18. Evaluation of DL models are performed

Learning Outcome 1: Exercise Deep Learning (DL)

Assessment Criteria	1. Fundamentals of DL is interpreted 2. Activities for DL Tools and libraries are performed 3. Data preparation is implemented 4. Manual and automatic data labeling are interpreted 5. Automatic labeling techniques are implemented
Conditions and Resources	1. Real or simulated workplace 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil, Eraser 7. Internet facilities 8. White board and marker 9. Audio Video Device
Contents	1 Fundamentals of DL 2 Activities for DL Tools and libraries 3 Data preparation 4 Manual and automatic data labeling 5 Automatic labeling techniques
Activities/job/Task	• In a customer support system, automatically label incoming emails as "urgent" if they contain specific keywords or phrases. • Use active learning to train a text classifier, where the model asks for human feedback on challenging samples. • Train a language model to predict missing words in a sentence, and then use the learned embeddings for downstream tasks. • Use a pre-trained image classification model on a large dataset and fine-tune it for a specific set of related images. • Apply clustering algorithms to group similar articles, then label all articles within a cluster with the dominant topic. • Classify a bag of images representing a scene, and the bag is labeled with the scene category. • Train a text classifier on a small labeled dataset and a large unlabeled dataset, improving performance with limited labeled data. • Automatically extract labels from social media posts or user-generated content using heuristics or pattern matching. • Train an initial text classifier, use it to label more data,

	and then retrain the model on the expanded dataset. • Train multiple classifiers and aggregate their predictions to assign labels to instances. In a customer support system, automatically label incoming emails as "urgent" if they contain specific keywords or phrases. • Use active learning to train a text classifier, where the model asks for human feedback on challenging samples. • Train a language model to predict missing words in a sentence, and then use the learned embeddings for downstream tasks. • Use a pre-trained image classification model on a large dataset and fine-tune it for a specific set of related images. • Apply clustering algorithms to group similar articles, then label all articles within a cluster with the dominant topic. • Classify a bag of images representing a scene, and the bag is labeled with the scene category. • Train a text classifier on a small labeled dataset and a large unlabeled dataset, improving performance with limited labeled data. • Automatically extract labels from social media posts or user-generated content using heuristics or pattern matching. • Train an initial text classifier, use it to label more data, and then retrain the model on the expanded dataset. Train multiple classifiers and aggregate their predictions to assign labels to instances.
Training Methods	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Assessment Methods	Assessment methods may include but not limited to 1. Written Test 2. Demonstration 3. Oral Questioning 4. Portfolio

Learning Experience 1: Exercise Deep Learning (DL)

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Exercise Deep Learning (DL)'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Exercise Deep Learning (DL)"	2. Read Information sheet 1: Exercise Deep Learning (DL) 3. Answer Self-check 1: Exercise Deep Learning (DL) 4. Check your answer with Answer key 1: Exercise Deep Learning (DL)
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job/Task Sheet and Specification Sheet Task-Sheet 1.1: In a customer support system, automatically label incoming emails as "urgent" if they contain specific keywords or phrases Task-Sheet 1.2: Use active learning to train a text classifier, where the model asks for human feedback on challenging samples Task-Sheet 1.3: Train a language model to predict missing words in a sentence, and then use the learned embeddings for downstream tasks. Task-Sheet 1.4: Use a pre-trained image classification model on a large dataset and fine-tune it for a specific set of related images. Task-Sheet 1.5: Apply clustering algorithms to group similar articles, then label all articles within a cluster with the dominant topic. Task-Sheet 1.6: Classify a bag of images representing a scene, and the bag is labeled with the scene category. Task-Sheet 1.7: Train a text classifier on a small labeled dataset and a large unlabeled dataset, improving performance with limited labeled data. Task-Sheet 1.8: Automatically extract labels from social media posts or user-generated

	content using heuristics or pattern matching. Task-Sheet 1.9: Train an initial text classifier, use it to label more data, and then retrain the model on the expanded dataset. Task-Sheet 1.10: Train multiple classifiers and aggregate their predictions to assign labels to instances.

Information Sheet 1: Exercise Deep Learning (DL)

Learning Objective:

After completion of this information sheet, the learners will be able to explain, define and interpret the following contents:

1.1 Interpret Fundamentals of DL

1.2 Perform Activities for DL Tools and libraries

1.3 Implement Data preparation

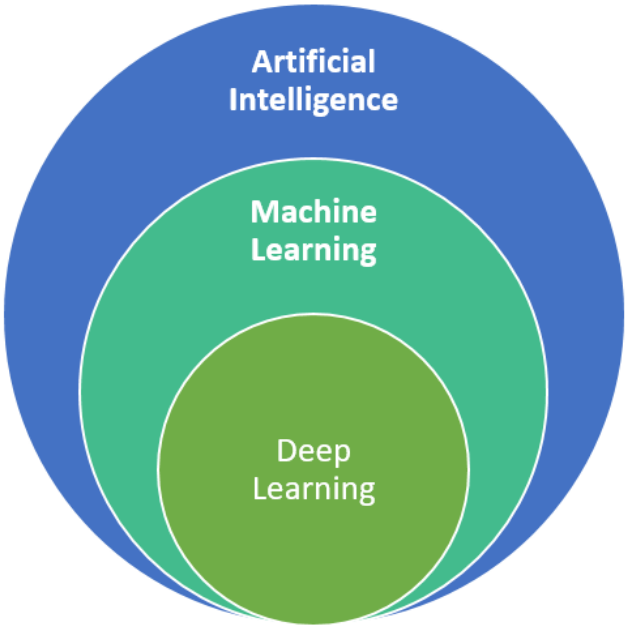
1.4 Interpret Manual and automatic data labeling

1.5 Implement Automatic labeling techniques

1.1. Interpret Fundamentals of Deep Learning (DL)

Definition:

Deep Learning (DL) is a subset of machine learning that uses neural networks with many layers to analyze and model complex data representations. The term "deep" refers to the multiple layers in neural networks that allow them to model highly abstract features and capture patterns in large amounts of data.



Key Concepts:

1. **Neural Networks:** Neural networks are composed of layers of nodes (neurons) that transform input data through mathematical functions to produce an output. These nodes are connected by weights that adjust during training.
2. **Layers in Neural Networks:**
    - **Input Layer:** Accepts input data (e.g., images, text).
    - **Hidden Layers:** Perform computations on the input data and learn various features.
    - **Output Layer:** Produces the final result (e.g., class predictions).
3. **Activation Functions:** Activation functions like ReLU (Rectified Linear Unit) or Sigmoid transform the input data in each neuron to introduce non-linearity and enable the model to learn complex patterns.
4. **Loss Functions:** Loss functions such as **Mean Squared Error (MSE)** or **Cross-Entropy** measure how well the model's predictions match the actual labels. The goal is to minimize the loss during training.

Example:

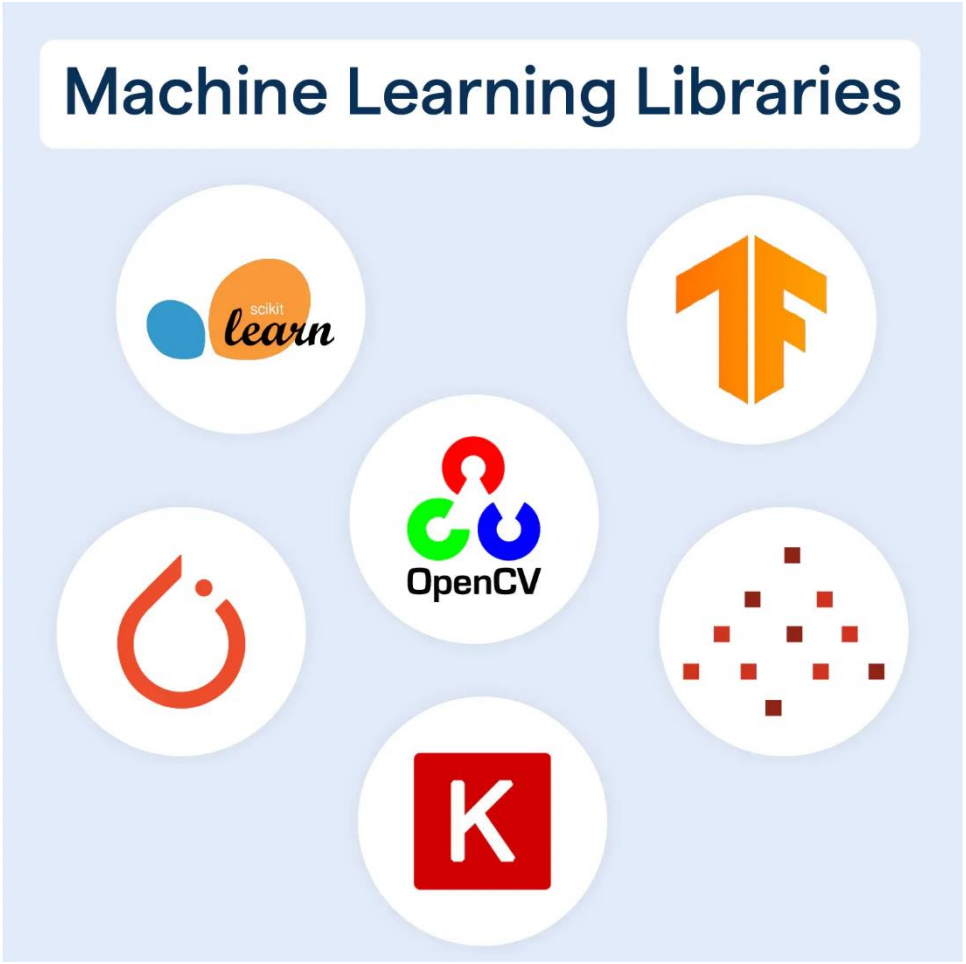
A simple neural network for classifying handwritten digits from the MNIST dataset contains:

- **Input Layer:** Each neuron represents a pixel in the image.
- **Hidden Layers:** Perform computations to extract features.
- **Output Layer:** Predicts the digit class (0-9).

1.2. Perform Activities for DL Tools and Libraries

Definition:

Deep learning frameworks and libraries provide the necessary tools and functionalities to build, train, and deploy deep learning models. These libraries abstract lower-level operations, making it easier to create sophisticated models.



Common DL Tools and Libraries:

1. **TensorFlow:** A widely used open-source framework developed by Google. It supports building and training deep learning models with scalability and flexibility.
2. **Keras:** A high-level API for building neural networks, typically used with TensorFlow or other backends. It simplifies the creation of deep learning models.
3. **PyTorch:** An open-source machine learning framework developed by Facebook, known for its dynamic computation graph and user-friendly interface, commonly used in research.
4. **OpenCV:** A library used primarily for computer vision tasks such as image processing, video analysis, and real-time processing.
5. **scikit-learn:** A popular machine learning library that can be used alongside deep learning tools for data preprocessing and evaluation.

Example Code (TensorFlow/Keras):

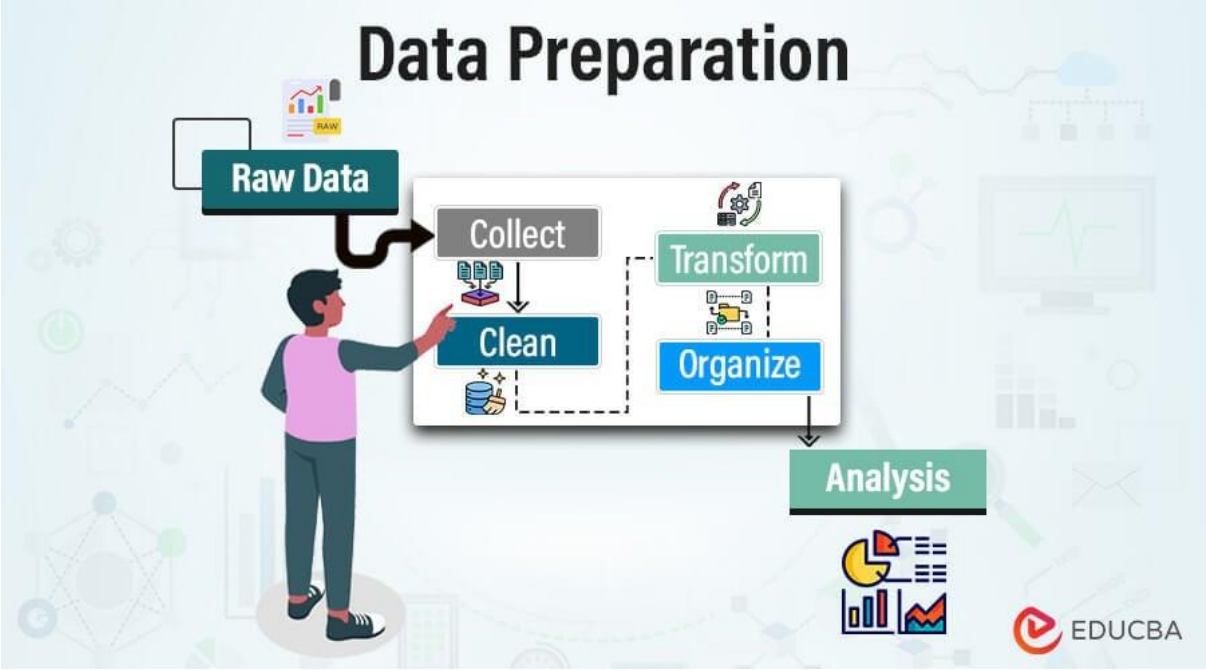
```
import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense
```

```
# Build a simple neural network model using Keras
model = Sequential([
    Dense(64, activation='relu', input_shape=(784,)),  # Input layer with 784 features (28x28 for MNIST)
    Dense(10, activation='softmax')  # Output layer for 10 classes (digits 0-9)
])

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

# Print model summary
model.summary()
```

1.3. Implement Data Preparation

Definition:

Data preparation involves transforming raw data into a format that is suitable for training deep learning models. This stage includes data collection, cleaning, normalization, augmentation, and splitting into training and testing sets.



Key Steps in Data Preparation:

1. **Data Collection:** Gathering the relevant data from different sources such as images, text, or sensors.
2. **Data Cleaning:** Removing irrelevant or noisy data, handling missing values, and standardizing the dataset to ensure consistency.
3. **Data Normalization/Standardization:** Scaling data to ensure all features are on the same scale. This helps deep learning models converge faster.
4. **Data Augmentation:** For image data, this can involve techniques like rotation, flipping, or zooming to artificially increase the diversity of the dataset.

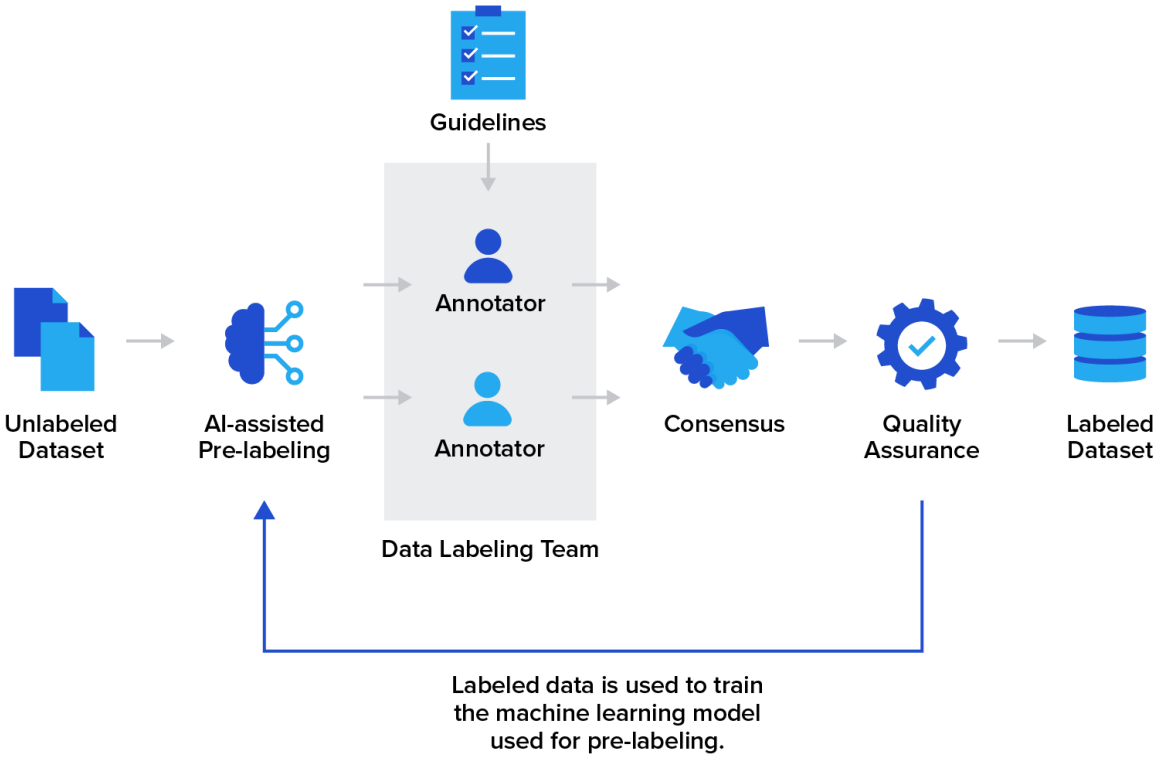
Example Code (Data Augmentation with Keras):

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

# Initialize the ImageDataGenerator with augmentation techniques
train_datagen = ImageDataGenerator(
    rescale=1./255,  # Normalize images to the range [0,1]
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)

# Load and augment images
train_generator = train_datagen.flow_from_directory(
    'data/train',  # Path to training images
    target_size=(64, 64),  # Resize images to 64x64
    batch_size=32,
    class_mode='categorical'  # For multi-class classification
)
```

1.4. Interpret Manual and Automatic Data Labeling

Definition:

Data labeling is the process of assigning labels (or classes) to raw data. In supervised learning, labeled data is essential for training models. Data labeling can either be manual or automatic.



Manual Data Labeling:

In manual labeling, human annotators assign labels to data. This process can be time-consuming but is necessary for tasks where human interpretation is crucial.

Example:

For image classification tasks, images might be stored in directories named by their class:

```
data/
    train/
        dog/
            dog1.jpg
            dog2.jpg
        cat/
```

cat1.jpg

cat2.jpg

Here, the images in the “dog” folder are labeled as "dog" and those in the “cat” folder as "cat."

Automatic Data Labeling:

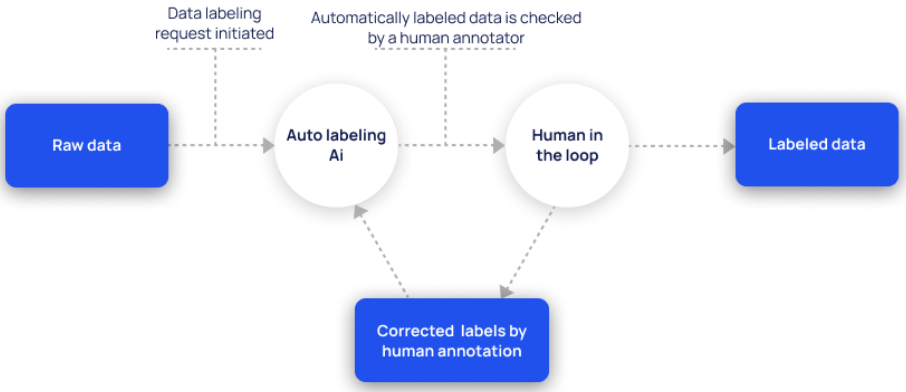
Automatic labeling uses pre-trained models or heuristics to assign labels without human intervention.

Example Code (Using Pre-trained ResNet50 for Automatic Labeling):

```
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import decode_predictions, preprocess_input

# Load a pre-trained ResNet50 model
model = ResNet50(weights='imagenet')

# Load and preprocess an image
img_path = 'image.jpg'
img = image.load_img(img_path, target_size=(224, 224))
img_array = image.img_to_array(img)
img_array = preprocess_input(img_array)

# Predict the label of the image
predictions = model.predict(img_array.reshape(1, 224, 224, 3))
decoded_predictions = decode_predictions(predictions, top=3)[0]

# Display predictions
for pred in decoded_predictions:
    print(f"Predicted label: {pred[1]} with probability {pred[2]}")
```

1.5. Implement Automatic Labeling Techniques

Definition: Automatic labeling techniques are methods used to label large datasets without requiring manual annotation. These techniques typically rely on pre-trained models or active learning algorithms.



Common Automatic Labeling Techniques:

1. **Transfer Learning:** Fine-tuning pre-trained models on a smaller dataset and using them to label new data.
2. **Active Learning:** A process in which a model selects uncertain samples for labeling, improving the model's performance iteratively.
3. **Semi-supervised Learning:** Uses a small amount of labeled data and a large amount of unlabeled data for training, thus reducing the reliance on large labeled datasets.

Example (Active Learning):

```
# Pseudo-code for active learning
# 1. Train an initial model on labeled data
model = train_initial_model_on_labeled_data()

# 2. Use the model to find uncertain samples from unlabeled data
uncertain_data = find_uncertain_samples(model, unlabeled_data)

# 3. Human annotators label the uncertain samples
labeled_data = human_labeling(uncertain_data)

# 4. Retrain the model with the newly labeled data
model = retrain_model_with_additional_labels(model, labeled_data)
```

Self-Check Sheet 1: Exercise Deep Learning

1. **What is Deep Learning (DL)?**
   Answer:

2. **What is the role of activation functions in neural networks?**
   Answer:

3. **What is the purpose of data augmentation in deep learning?**
   Answer:

4. **What is the difference between manual and automatic data labeling?**
   Answer:

5. **What is Active Learning in the context of automatic labeling techniques?**
   Answer:

6. **What is the function of a loss function in deep learning models?**

   Answer:

7. **How does transfer learning help in automatic labeling?**

   Answer:

8. **What are some commonly used deep learning frameworks?**

   Answer:

9. **What is data normalization and why is it important in deep learning?**

   Answer:

10. **What is the role of a pre-trained model like ResNet50 in automatic labeling?**

    Answer:

Answer Sheet 1: Exercise Deep Learning

1. **What is Deep Learning (DL)?**
   Answer: Deep Learning (DL) is a subset of Machine Learning (ML) that uses neural networks with multiple layers to model complex patterns in large amounts of data, allowing it to learn high-level abstractions in data.

2. **What is the role of activation functions in neural networks?**
   Answer: Activation functions, such as ReLU and Sigmoid, help neural networks model complex functions by introducing non-linearity to the output of neurons, enabling the network to learn from data more effectively.

3. **What is the purpose of data augmentation in deep learning?**
   Answer: Data augmentation is used to artificially expand the dataset by applying transformations like rotation, flipping, or cropping, which helps improve the robustness and generalization of deep learning models.

4. **What is the difference between manual and automatic data labeling?**
   Answer: Manual data labeling involves human annotators assigning labels to data, while automatic data labeling uses pre-trained models or algorithms to assign labels without human intervention, saving time and resources.

5. **What is Active Learning in the context of automatic labeling techniques?**
   Answer: Active Learning is a technique where a model iteratively queries the most uncertain data points for labeling by a human, helping to improve the model's performance with fewer labeled samples.

6. **What is the function of a loss function in deep learning models?**

   Answer: The loss function measures how well the model's predictions match the actual values. It guides the optimization process by providing a metric for minimizing prediction errors. Common examples include Mean Squared Error (MSE) and Cross-Entropy.

7. **How does transfer learning help in automatic labeling?**

   Answer: Transfer learning allows a pre-trained model to be fine-tuned on a new dataset, enabling it to automatically label new data based on learned features from a related task, reducing the need for manual annotation.

8. **What are some commonly used deep learning frameworks?**

   Answer: Some commonly used deep learning frameworks include TensorFlow, Keras, PyTorch, OpenCV, and scikit-learn, each offering tools for building, training, and deploying deep learning models.

9. **What is data normalization and why is it important in deep learning?**

   Answer: Data normalization involves scaling features so they lie within a similar range, typically [0, 1]. It helps speed up the training process and ensures faster convergence by preventing certain features from dominating others.

10. **What is the role of a pre-trained model like ResNet50 in automatic labeling?**

    Answer: A pre-trained model like ResNet50 can be used for automatic labeling by making predictions on new data, such as images, and generating labels based on its learned knowledge from large datasets like ImageNet. This eliminates the need for manual labeling.

Task-Sheet 1.1: In a customer support system, automatically label incoming emails as "urgent" if they contain specific keywords or phrases

Steps

Step 1: Understand the Problem

- The goal is to label incoming customer support emails as "urgent" based on keywords or phrases that indicate urgency (e.g., "immediate help", "critical", "emergency", etc.).
- This will help categorize and prioritize customer requests for more efficient handling.

Step 2: Collect and Prepare Data

- Gather a set of sample emails from the customer support system.
- Manually label some emails as "urgent" or "not urgent" based on the presence of keywords.

Step 3: Define Keywords and Phrases

- Identify and define a list of keywords and phrases that indicate urgency (e.g., "urgent", "ASAP", "immediate", "critical").
- Optionally, you can use NLP techniques to further define context-specific urgency.

Step 4: Text Preprocessing

- Tokenize the text in each email to split it into words.
- Convert the text to lowercase to ensure uniformity.
- Remove any unnecessary punctuation or special characters.

Step 5: Implement a Keyword Matching Algorithm

- Create an algorithm that checks each email for the presence of any of the predefined "urgent" keywords or phrases.
- If a match is found, label the email as "urgent".
- If no match is found, label the email as "not urgent".

Step 6: Model Evaluation and Testing

- Test the system using a set of unseen emails to check if the classification is accurate.
- Evaluate the performance by measuring the number of false positives and false negatives.

Step 7: Integrate into Customer Support System

- Once the email classification system is working, integrate it into the customer support platform to automatically categorize new incoming emails.

Step 8: Fine-Tuning and Improvement

- Continuously update the list of keywords based on feedback and additional analysis.
- Optionally, integrate machine learning for context-aware classification if keyword-based methods become insufficient.

Task-Sheet 1.2: Use active learning to train a text classifier, where the model asks for human feedback on challenging samples

Steps

Step 1: Understand Active Learning

- Active learning is a semi-supervised machine learning technique where the model queries the human annotator for labels on the most informative or uncertain samples.
- This is useful in situations where labeling large datasets is costly or time-consuming, as it reduces the number of samples requiring manual annotation.

Step 2: Define the Problem and Dataset

- Choose a text classification task (e.g., spam detection, sentiment analysis, etc.).
- Collect or create a dataset for this task. The dataset should contain labeled and unlabeled samples.
- Define the classes that the classifier will predict.

Step 3: Preprocess the Text Data

- Clean the text data by:
    - Lowercasing all text.
    - Removing punctuation, special characters, and stop words.
    - Tokenizing the text (splitting it into individual words or subwords).
    - Vectorizing the text using methods like TF-IDF, Word2Vec, or embeddings.

Step 4: Initialize the Classifier

- Train a basic model on a small initial labeled dataset. This will serve as the starting point for active learning.
- Popular classifiers include logistic regression, support vector machines (SVM), or deep learning models like RNNs or transformers.

Step 5: Implement Active Learning Loop

- Identify uncertain or ambiguous samples in the dataset using a strategy like uncertainty sampling, where the model queries the samples for which it is least confident.
- Train the model iteratively, with each cycle consisting of:
    1. The model predicts labels for a batch of unlabeled samples.
    2. The model requests human feedback for the most uncertain predictions.
    3. The human annotator labels the samples.
    4. Add these labeled samples to the training set.
    5. Retrain the model on the updated dataset.

Step 6: Optimize the Query Strategy

- Try different active learning strategies such as:
    - Uncertainty sampling (model queries the samples it is least confident about).
    - Query by committee (multiple models vote on which samples should be labeled).
    - Diversity-based sampling (samples selected to cover a diverse range of the feature space).

Step 7: Evaluate the Model

- Evaluate the classifier's performance using standard metrics like accuracy, precision, recall, and F1-score on a held-out test set.
- Track the number of human labels used and the performance improvement with each iteration.

Step 8: Fine-Tuning and Model Improvement

- Fine-tune the model based on feedback and performance evaluation.
- You may experiment with different text processing techniques, model architectures, and active learning strategies to further improve accuracy.

Task-Sheet 1.3: Train a language model to predict missing words in a sentence, and then use the learned embeddings for downstream tasks.

Steps

Step 1: Understand the Task

- The task involves training a language model to perform **masked language modeling (MLM)**. MLM is a pretraining task used by models like BERT, where some words in a sentence are masked, and the model must predict the missing words.
- The embeddings learned by the model during training can later be used for downstream tasks such as text classification, sentiment analysis, and named entity recognition (NER).

Step 2: Choose a Language Model Architecture

- You will train a **Transformer-based model** like BERT, GPT, or RoBERTa. These models are designed to work with large text datasets and learn contextual relationships between words.
- For this task, we'll use a pre-trained model like **BERT** and fine-tune it on the MLM task, but you can also explore training the model from scratch with a large corpus.

Step 3: Prepare the Dataset

- Choose a large text corpus for training (e.g., Wikipedia, books, or news articles).
- Preprocess the data to tokenize the text and create a training dataset with masked words. Tools like **Hugging Face's Tokenizers** library can help in this process.
- The dataset should include sentences where a certain percentage of words are randomly masked, and the model needs to predict these masked words.

Step 4: Preprocess and Tokenize Text

- Tokenize the sentences into words or subword tokens using tokenizers like **WordPiece** or **Byte Pair Encoding (BPE)**.
- Mask a percentage of words (e.g., 15%) in each sentence randomly, following the standard approach used by BERT.
- Convert text into input IDs and attention masks, which are needed for Transformer models.

Step 5: Train the Model

- Fine-tune a pre-trained language model (e.g., BERT) using the masked language modeling task.
    - Use the **Hugging Face Transformers** library to load a pre-trained BERT model and tokenizer.
    - Define the loss function (cross-entropy loss) for masked word prediction.
    - Use an optimizer like **AdamW** and fine-tune the model on your prepared dataset.

Step 6: Train the Model with the Data

- Train the model using the prepared dataset. Use GPU acceleration (e.g., with **CUDA** or cloud services like **Google Colab** or **AWS**).
- Set the appropriate batch size, learning rate, and the number of epochs for training.

Step 7: Save the Learned Embeddings

- After training, extract the word embeddings learned by the model. These embeddings are dense vector representations of words and can be used for downstream tasks.
- Save the model's weights and embeddings to disk for future use.

Step 8: Use Learned Embeddings for Downstream Tasks

- Once the model has been trained, use the learned embeddings to:
    - Perform **text classification**: Use the embeddings to classify text into categories (e.g., spam vs. not spam).
    - Perform **sentiment analysis**: Use the embeddings to predict the sentiment of text (positive, negative, neutral).
    - Perform **named entity recognition (NER)**: Use the embeddings to identify named entities like people, locations, and organizations in text.

Step 9: Evaluate the Model

- Evaluate the language model by checking its ability to predict missing words.
- For downstream tasks, fine-tune the model further with labeled data for the specific task (e.g., sentiment analysis or text classification).
- Use metrics like **accuracy**, **precision**, **recall**, and **F1-score** for task evaluation.

Task-Sheet 1.4: Use a pre-trained image classification model on a large dataset and fine-tune it for a specific set of related images.

Steps

Step 1: Select a Pre-trained Model

- Choose a pre-trained image classification model that has been trained on a large dataset like **ImageNet**.
- Models like **ResNet**, **VGG**, **Inception**, or **EfficientNet** are widely used in image classification tasks.
- Load a pre-trained model from a framework like **PyTorch** or **TensorFlow**.

Step 2: Dataset Preparation

- **Collect Your Dataset**: Gather a specific dataset for your task. For example, if you're fine-tuning a model for animal classification, you might gather images of cats, dogs, and other animals.
- **Label Your Data**: Ensure the dataset has clear labels (e.g., cat, dog, bird) and is divided into training, validation, and test sets.
- **Preprocess Images**: Resize all images to the same size expected by the pre-trained model (e.g., 224x224 pixels for VGG/ResNet).
- **Augment Data**: Use techniques like flipping, rotation, and cropping to increase dataset size and variability (optional but recommended).

Step 3: Prepare Data for Model Input

- **Normalization**: Normalize image pixel values (e.g., scaling between 0 and 1 or standardizing to a mean and standard deviation for the pre-trained model).
- **Data Loading**: Use data loading tools in PyTorch or TensorFlow (e.g., torch.utils.data.DataLoader or tf.data.Dataset) to efficiently handle batches of data.

Step 4: Load the Pre-trained Model

- Load the pre-trained model and configure it for fine-tuning.
    - In **PyTorch**, this can be done using torchvision.models (e.g., models.resnet18(pretrained=True)).
    - In **TensorFlow**, you can load models like **EfficientNetB0** via tf.keras.applications.

Step 5: Modify the Final Layers

- Modify the last layer of the pre-trained model to match the number of classes in your dataset.
    - If the model is trained for 1000 classes (e.g., ImageNet), but you only need 3 (e.g., cat, dog, and bird), replace the last layer (fully connected layer) to have 3 output units.
- **Optional**: You may freeze the earlier layers (for feature extraction) and only train the final layers initially, or you can fine-tune the entire network.

Step 6: Fine-tune the Model

- **Select Loss Function**: For image classification, use a **cross-entropy loss** function.
- **Choose Optimizer**: Use an optimizer like **Adam** or **SGD**.

- **Train the Model**: Fine-tune the model on your dataset using your training data. Start by training the last few layers, then gradually unfreeze earlier layers for more fine-tuning.
    - Monitor the training and validation loss to avoid overfitting.
- **Epochs and Batch Size**: Choose a reasonable number of epochs (e.g., 10-20) and batch size (e.g., 32) based on your dataset size.

Step 7: Evaluate the Model

- After fine-tuning, evaluate the model's performance on the validation and test sets.
- Track metrics such as **accuracy**, **precision**, **recall**, and **F1-score**.
- If the model performs well, use it for inference on new images.

Step 8: Save and Deploy the Model

- **Save the Model**: Once training is complete, save the fine-tuned model for future use. In PyTorch, this can be done using torch.save(), and in TensorFlow, using model.save().
- **Deploy the Model**: Use the saved model to make predictions on new images or deploy it into a production system.

Task-Sheet 1.5: Apply clustering algorithms to group similar articles, then label all articles within a cluster with the dominant topic.

Steps

Step 1: Collect and Preprocess Articles

- **Collect Data**: Gather a collection of articles to be clustered. These can be news articles, research papers, or any other text-based content.
- **Preprocess Text Data**:
    - Tokenize the articles into words.
    - Remove stop words (e.g., "the", "is", "and").
    - Apply lemmatization/stemming to normalize words.
    - Remove non-textual content (e.g., special characters, URLs).

Step 2: Vectorize the Articles

- **Convert text to numerical features**: Use text vectorization techniques like **TF-IDF (Term Frequency-Inverse Document Frequency)** or **Word2Vec** to represent the articles as numerical vectors.
    - In **TF-IDF**, the importance of words is calculated based on how frequently they appear in the document and across all documents.
    - **Word2Vec** maps words to dense vectors that capture semantic meanings and relationships between words.
- **Libraries**: Use libraries such as **Scikit-learn** or **Gensim** for text vectorization.

Step 3: Apply Clustering Algorithms

- **Select Clustering Algorithm**:
    - Choose an appropriate clustering algorithm for grouping the articles. Common options include:
        - **K-Means Clustering**: Works well for large datasets and predefined cluster counts.
        - **DBSCAN (Density-Based Spatial Clustering of Applications with Noise)**: Can handle clusters of arbitrary shapes and is useful when you don’t know the number of clusters in advance.
        - **Agglomerative Clustering**: A hierarchical method that builds a tree of clusters.
    - Fit the model to your vectorized article data and specify the number of clusters or other hyperparameters as needed.

Step 4: Analyze and Label Clusters

- **Analyze Each Cluster**: After clustering, examine the words or features that are most prevalent in each cluster. This can be done by:
    - **Top N terms**: Extract the most common terms in each cluster by looking at the most frequent words in the TF-IDF matrix for each group.
    - **Topic Modeling (Optional)**: Use **Latent Dirichlet Allocation (LDA)** or **Non-Negative Matrix Factorization (NMF)** to automatically discover topics in each cluster based on word distributions.

- **Assign Labels**: Based on the most frequent terms or the topics identified, assign a dominant topic label to each cluster (e.g., "Technology", "Healthcare", "Politics").

Step 5: Evaluate the Clustering Performance

- **Cluster Validation**: Evaluate the clustering performance by checking metrics like:
    - **Silhouette Score**: Measures how similar an article is to its own cluster compared to other clusters.
    - **Adjusted Rand Index (ARI)**: Measures the similarity between two data clusterings.
- **Visualization**: Use dimensionality reduction techniques (e.g., **t-SNE** or **PCA**) to visualize the clusters in 2D or 3D space for better interpretation.

Step 6: Interpret Results and Fine-Tune

- Review the clusters and their labels to ensure that the articles within a cluster are indeed related to the assigned dominant topic.
- If the clusters are not satisfactory, adjust the clustering parameters (e.g., change the number of clusters in K-Means or DBSCAN's epsilon value) and re-run the algorithm.

Step 7: Save and Deploy

- **Save the Model**: Save the clustering model and the vectorizer to deploy them later. In **Scikit-learn**, you can use **joblib** or **pickle** to save the model and vectorizer.
- **Deploy for Real-time Classifications**: If needed, deploy the clustering model to label new articles dynamically.

Task-Sheet 1.6: Classify a bag of images representing a scene, and the bag is labeled with the scene category.

Steps

Step 1: Gather and Prepare the Dataset

- **Dataset Collection**: Obtain a dataset containing a collection of labeled images, where each label corresponds to a scene category. For example, you can use a dataset like **Places365** or create your own collection.
- **Preprocessing**:
    - Resize images to a standard shape (e.g., 224x224 pixels).
    - Normalize pixel values to a range of [0, 1] or [-1, 1] for model compatibility.
    - Apply augmentation techniques such as random rotation, flipping, and zooming to increase the dataset diversity.

Step 2: Organize the Data

- **Bag of Images**: Organize the images into bags, where each bag represents a single scene. Each scene may consist of multiple images that contribute to the overall scene classification. Label each bag with the corresponding scene category.
- **Split Data**: Divide the dataset into training, validation, and test sets to evaluate model performance.

Step 3: Choose a Model Architecture

- **Model Selection**:
    - Use a **Convolutional Neural Network (CNN)** for image feature extraction. A popular architecture is **ResNet**, **VGGNet**, or **Inception**. If computational power is available, consider using a pre-trained model and fine-tuning it for your task.
    - **Bag-Level Classification**: Since you are classifying a "bag of images", you may need to aggregate features from all images in the bag. One approach is to use **feature pooling** (mean or max pooling) over the features extracted from individual images.

Step 4: Implement the Model

- **Input Layer**: Accept a bag of images (multiple images per scene).
- **Feature Extraction**: Use a CNN to extract features from each image.
    - For each image, use a pre-trained CNN model or a custom CNN to extract feature maps.
    - Aggregate the features from the multiple images in the bag to form a unified feature representation for the entire scene.
- **Fully Connected Layer**: After feature aggregation, pass the result through a fully connected layer(s) to classify the scene category.
- **Output Layer**: The final output layer should have as many neurons as the number of scene categories (e.g., 10 categories → 10 neurons).

Step 5: Train the Model

- **Loss Function**: Use **categorical cross-entropy loss** for multi-class classification tasks.
- **Optimization**: Use **Adam** or **SGD** for optimization.

- **Metrics**: Track accuracy, precision, recall, and F1-score.
- **Training Process**: Train the model using the training data and validate it using the validation set. Ensure to use early stopping to avoid overfitting.

Step 6: Evaluate the Model

- **Performance Evaluation**: After training the model, evaluate its performance on the test set.
- **Confusion Matrix**: Create a confusion matrix to visualize classification errors across different categories.
- **Fine-tuning**: Based on the results, you can fine-tune the model by adjusting hyperparameters, such as learning rate, batch size, or the number of layers in the CNN.

Step 7: Model Deployment

- **Save the Model**: Save the trained model using **TensorFlow/Keras** or **PyTorch** (depending on the framework used) for deployment.
- **Deploy for Inference**: Deploy the model to classify new bags of images representing scenes, allowing the system to predict the category based on unseen data.

Task-Sheet 1.7: Train a text classifier on a small labeled dataset and a large unlabeled dataset, improving performance with limited labeled data.

Steps

Step 1: Data Collection and Preprocessing

1. **Gather Data**:
    - **Labeled Data**: Collect a small labeled dataset (e.g., 1,000-5,000 samples) for training.
    - **Unlabeled Data**: Collect a large corpus of unlabeled text data related to the classification task (e.g., several tens of thousands of samples).
2. **Data Preprocessing**:
    - **Text Cleaning**: Remove unnecessary characters, punctuation, and perform tokenization.
    - **Text Normalization**: Convert text to lowercase, remove stop words, and apply stemming/lemmatization.
    - **Vectorization**: Convert text data into numerical format using techniques like **TF-IDF** or **Word2Vec** embeddings, or use pre-trained embeddings like **GloVe** or **BERT** for improved performance.
3. **Dataset Splitting**:
    - Split the labeled data into **training** and **validation** sets (e.g., 80%-20% split).
    - Keep the unlabeled data aside to be utilized for semi-supervised learning.

Step 2: Model Selection

1. **Choose Model Architecture**:
    - Use a **pre-trained language model** like **BERT** or **GPT-3** to leverage transfer learning for text classification. This can be further fine-tuned on the small labeled dataset.
    - Alternatively, if using simpler models, you could start with **Logistic Regression** or **Naive Bayes** with **TF-IDF** features for the baseline model.
2. **Semi-Supervised Learning**:
    - Use techniques like **self-training**, where the model is trained on the labeled data and then iteratively labels the unlabeled data, improving with each iteration.
    - Alternatively, implement **pseudo-labeling**, where the model generates predictions for the unlabeled data, and high-confidence predictions are added to the training set as pseudo-labels.

Step 3: Model Training and Evaluation

1. **Train the Model on the Labeled Data**:
    - Fine-tune a pre-trained model on the small labeled dataset or train a simple model like **Logistic Regression** using the TF-IDF features.
    - For deep learning models, use **Adam** optimizer and **categorical cross-entropy** loss for multi-class classification tasks.

2. **Use Unlabeled Data for Semi-Supervised Learning**:
    - After initial training, apply **pseudo-labeling** or **self-training** to label the unlabeled dataset. Add the newly labeled data into the training set to improve the model iteratively.
    - You can also use **active learning** to selectively query human annotations for the most uncertain predictions.
3. **Model Evaluation**:
    - Evaluate the model using **accuracy**, **precision**, **recall**, and **F1-score** on the validation set.
    - Experiment with different evaluation metrics depending on the nature of the classification task (binary vs. multi-class).
    - Use **confusion matrices** to visualize classification performance and identify potential errors.

Step 4: Model Improvement

1. **Data Augmentation**:
    - Use techniques like **synonym replacement** or **text paraphrasing** to augment the labeled dataset. This will artificially increase the amount of labeled data, improving the model's performance.
2. **Fine-tuning**:
    - Fine-tune hyperparameters such as the learning rate, batch size, and the number of layers/neurons in the neural network to improve performance.
3. **Regularization**:
    - Implement dropout, batch normalization, or L2 regularization to avoid overfitting, especially when using a small labeled dataset.

Step 5: Model Deployment

1. **Model Export**:
    - Save the trained model in a deployable format (e.g., TensorFlow .h5 or PyTorch .pth format).
2. **Deploy**:
    - Deploy the model to classify incoming text data in a live environment (e.g., an API for real-time classification).

Task-Sheet 1.8: Automatically extract labels from social media posts or user-generated content using heuristics or pattern matching.

Steps

Step 1: Data Collection and Preprocessing

1. **Collect Data**:
    - **Social Media Posts**: Gather a collection of social media posts or user-generated content from platforms such as Twitter, Reddit, or forums. This data can be collected through APIs like Twitter API or web scraping techniques (ensure you comply with data usage policies).
    - **Labeling Criteria**: Define a set of labels or categories that you want to extract from the content (e.g., "Product Feedback", "Customer Complaints", "General Inquiry").
2. **Text Cleaning**:
    - Remove unnecessary characters like URLs, hashtags, emojis, and special symbols.
    - Convert text to lowercase to maintain consistency and avoid case-sensitive mismatches.
    - Perform tokenization to split text into words or phrases for easier pattern matching.
3. **Data Preprocessing**:
    - **Stopword Removal**: Remove common words that do not contribute to the meaning (e.g., "the", "is", "and").
    - **Stemming/Lemmatization**: Reduce words to their root form to enhance pattern matching (e.g., "running" → "run").

Step 2: Heuristic or Pattern Matching Techniques

1. **Define Label Extraction Patterns**:
    - Identify specific keywords or phrases associated with each label. For example:
        - For "Product Feedback": Look for keywords like "love", "hate", "recommend", "like", "dislike".
        - For "Customer Complaints": Look for terms such as "problem", "issue", "disappointed", "refund", "broken".
        - For "General Inquiry": Look for terms like "how", "why", "what", "please explain", "can you help".
2. **Pattern Matching**:
    - Use regular expressions (regex) to create rules that match these patterns in the text. This can help identify specific keywords or phrases that correspond to each label.
    - You can also use **string matching techniques** to search for exact or approximate matches of keywords.
3. **Heuristics for Classification**:

- Based on the identified patterns, create simple heuristics to classify posts into predefined categories. For example:
  - If a post contains words like "disappointed", "refund", and "issue", classify it as "Customer Complaints".
  - If a post contains "love", "recommend", classify it as "Product Feedback".

Step 3: Implementing the Label Extraction

1. **Create a Function to Extract Labels**:
   - Write a function to iterate over the text data and apply the heuristics or patterns.
   - For each post, use the predefined rules to check for the presence of specific keywords and assign the corresponding label to the post.
2. **Handling Multiple Labels**:
   - For posts that may contain multiple labels (e.g., a post that expresses both positive feedback and a complaint), you can assign multiple labels based on the matches.
3. **Refinement**:
   - Review a subset of the results to ensure accuracy and adjust the heuristics or pattern matching rules if necessary.

Step 4: Testing and Evaluation

1. **Evaluation**:
   - Evaluate the accuracy of the labeling process by manually reviewing a sample of the posts and comparing the automatically extracted labels with human annotations.
   - Measure metrics like **precision**, **recall**, and **F1-score** to assess the effectiveness of the extraction process.
2. **Refinement**:
   - Based on the evaluation, refine the rules, patterns, or heuristics to improve accuracy.
   - Consider expanding the pattern matching to include more variations or edge cases.

Step 5: Final Deployment

1. **Model or Script Deployment**:
   - Once the labeling process is refined and performs well, the script or system can be deployed to continuously monitor incoming social media posts or user-generated content.
2. **Automation**:
   - Automate the task to run at scheduled intervals or in real-time using webhooks or API integrations to process new data as it becomes available.

Task-Sheet 1.9: Train an initial text classifier, use it to label more data, and then retrain the model on the expanded dataset.

Steps

Step 1: Initial Text Classifier Training

1. **Prepare the Initial Dataset**:
    - Collect a labeled dataset that can be used to train the initial text classifier. This dataset should consist of text data (e.g., emails, product reviews, news articles) along with predefined labels (e.g., sentiment labels, topic categories).
2. **Preprocess the Text Data**:
    - **Text Cleaning**: Remove unnecessary characters (e.g., punctuation, stop words, emojis).
    - **Tokenization**: Split the text into words or phrases.
    - **Vectorization**: Convert the text into numerical format using methods like **TF-IDF**, **CountVectorizer**, or **word embeddings**.
3. **Choose a Classification Model**:
    - Choose an appropriate model for text classification such as **Logistic Regression**, **Naive Bayes**, **Support Vector Machine (SVM)**, or **Deep Learning Models** (e.g., **RNN**, **CNN**, **BERT**).
4. **Train the Classifier**:
    - Train the selected model using the preprocessed labeled data. Monitor the training process and assess the performance using validation techniques like **cross-validation**.
5. **Evaluate Initial Model**:
    - Evaluate the performance of the trained classifier using metrics such as **accuracy**, **precision**, **recall**, and **F1-score** on a test set (unseen data).

Step 2: Use the Trained Model to Label More Data

1. **Select Unlabeled Data**:
    - Choose a dataset that has not been labeled. This could be new incoming data (e.g., recent customer reviews, social media posts) or a larger pool of unlabeled text data.
2. **Label the Unlabeled Data**:
    - Use the trained model to predict labels for the unlabeled data. This is the **pseudo-labeling** step where the model automatically assigns labels based on its current understanding.
3. **Review and Filter Predicted Labels**:
    - Review a sample of the pseudo-labeled data to ensure that the model's predictions are reasonable and accurate. You can use human annotators to validate and correct any incorrect predictions if needed.
4. **Expand the Dataset**:

- Combine the newly labeled data with the original labeled dataset to form an expanded training dataset.

Step 3: Retrain the Model on the Expanded Dataset

1. **Preprocess the Expanded Dataset**:
   - Preprocess both the original and newly labeled data using the same preprocessing steps (text cleaning, tokenization, and vectorization) to ensure consistency in the input format.
2. **Retrain the Classifier**:
   - Train the classifier again using the expanded dataset. This can be done by fine-tuning the model, adjusting hyperparameters, or retraining from scratch depending on the size and quality of the data.
3. **Evaluate the Retrained Model**:
   - Evaluate the performance of the retrained classifier using the same evaluation metrics as in Step 1. Compare the performance of the initial model with the retrained model to see if there is an improvement.
4. **Monitor Overfitting**:
   - Ensure that the retraining process does not lead to overfitting by monitoring performance on a validation set and applying techniques like **early stopping** or **regularization** if necessary.

Step 4: Iterative Process and Model Improvement

1. **Repeat the Process**:
   - If necessary, repeat the process by using the retrained model to label more data and retrain again. This iterative process can help improve the model incrementally.
2. **Improve Model Performance**:
   - Explore advanced techniques like **semi-supervised learning** or **active learning** to further enhance the model's performance with limited labeled data.
3. **Deploy the Model**:
   - Once the model reaches satisfactory performance, deploy it to classify new text data automatically in production environments.

Task-Sheet 1.10: Train multiple classifiers and aggregate their predictions to assign labels to instances.

Steps

Step 1: Data Preparation

1. **Gather Dataset**:
    - Obtain a labeled dataset for training the classifiers. The dataset should consist of features (e.g., text, images, numerical data) and corresponding labels.
2. **Preprocess Data**:
    - **Text Data**: Clean and preprocess the text by removing stop words, punctuation, and irrelevant characters. Apply techniques like tokenization, lemmatization, or stemming.
    - **Numerical Data**: Normalize or standardize numerical features to bring them to a common scale.
    - **Categorical Data**: Encode categorical features using techniques such as one-hot encoding or label encoding.
3. **Split Data**:
    - Divide the dataset into training and validation sets (e.g., 80% training, 20% validation). Alternatively, use **k-fold cross-validation** for better generalization.

Step 2: Train Multiple Classifiers

1. **Choose Classifiers**:
    - Select a variety of classifiers to train, such as:
        - **Logistic Regression**
        - **Decision Trees**
        - **Support Vector Machine (SVM)**
        - **Random Forest**
        - **K-Nearest Neighbors (KNN)**
        - **Neural Networks** (e.g., MLP)
2. **Train Classifiers**:
    - Train each classifier on the training dataset. Tune the hyperparameters of each classifier using **GridSearchCV** or **RandomizedSearchCV** for optimal performance.
3. **Evaluate Performance**:
    - After training, evaluate each model's performance using validation data. Metrics such as **accuracy**, **precision**, **recall**, and **F1-score** can be used to assess individual model performance.

Step 3: Aggregate Predictions

1. **Ensemble Methods**:
    - Use an ensemble method to combine the predictions of the multiple classifiers:

        - **Majority Voting**: Each model votes on the predicted label, and the label with the majority votes is selected.
        - **Averaging**: For regression tasks, average the predictions of all models.
        - **Weighted Voting**: Each classifier's vote is weighted by its performance (e.g., models with higher accuracy get more influence).
2. **Train Aggregation Model**:
    - If necessary, you can train a meta-classifier (also known as a **stacking model**) that takes the predictions of the individual classifiers as input and learns to combine them into a final label.
    - This meta-model could be a **Logistic Regression** or **SVM**.

Step 4: Evaluate the Combined Model

1. **Assess Performance**:
    - Evaluate the performance of the combined model on the test set (or out-of-sample data).
    - Compare the performance of the ensemble method to the individual classifiers to check if the ensemble improves overall accuracy and robustness.
2. **Fine-Tune**:
    - If necessary, fine-tune the ensemble method or adjust the individual models to improve performance.

Step 5: Deploy the Model

1. **Deploy the Ensemble Model**:
    - Once satisfied with the performance, deploy the trained ensemble model for real-time or batch prediction.
2. **Monitor and Update**:
    - Continuously monitor the performance of the deployed model on new data, and periodically retrain the classifiers if performance degrades or if new data is available.

Learning Outcome 2: Analyze features

Assessment Criteria	1. Feature extraction is implemented 2. Visualization of word vectors are implemented with word cloud, histogram, heatmap, plots and tableau. 3. Embedding models are exercised 4. Pre-trained word embedding is implemented
Condition and Resource	1. Actual workplace or training environment 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil and Eraser 7. Internet Facilities 8. Whiteboard and Marker 9. Imaging Device (Digital camera, scanner etc.)
Content	▪ Visualization of word vectors with word cloud, histogram, heatmap, plots and tableau ▪ Embedding models are exercised o Word representation o Embedding matrix o Word2Vec, FastText and Glove ▪ Pre-trained word embedding is implemented o Implications of pre-trained word vectors o Tuning the word vectors o Embedding model evaluation (intrinsic and extrinsic)
Activity/ task/ Job	• Use a dataset of text (e.g., news articles or tweets) and create a word cloud to visualize the frequency of words. Experiment with adjusting parameters like word frequency and font size. • Analyze the distribution of word frequencies in a given text corpus using a histogram. Identify the most common and least common words. • Utilize pre-trained word embeddings (e.g., Word2Vec, GloVe) to create a heatmap representing the similarity between words. Choose a set of words and visualize their pairwise similarities. • Pick a set of words and plot their embeddings in a 2D or 3D scatter plot. Analyze the relationships and clusters among the words. • Build a Word2Vec model from scratch using a small text corpus. Train the model to generate

	word embeddings for words in the corpus. • Implement the training of a GloVe (Global Vectors for Word Representation) model using a given text dataset. Experiment with different hyperparameters. • Choose a pre-trained word embedding model (e.g., Word2Vec, GloVe, FastText) and use it to obtain word vectors for a set of words. Explore the semantic relationships. • Implement a function to calculate the similarity between two word vectors. Test it with words that should have similar and dissimilar meanings. Use word embeddings to build a sentiment analysis model. Train the model to classify text sentiment (positive, negative, neutral) using a dataset with labeled sentiments.
Training Technique	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Methods of Assessment	1. Written Test 2. Demonstration 3. Oral Questioning 4. Portfolio

Learning Experience 2: Analyze features

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Exercise Deep Learning (DL)'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Analyze features"	2. Read Information sheet 2: Analyze features 3. Answer Self-check 2: Analyze features 4. Check your answer with Answer key 1: Analyze features
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job/Task Sheet and Specification Sheet Task Sheet 2.1: Use a dataset of text (e.g., news articles or tweets) and create a word cloud to visualize the frequency of words. Experiment with adjusting parameters like word frequency and font size. Task Sheet 2.2: Analyze the distribution of word frequencies in a given text corpus using a histogram. Identify the most common and least common words. Task Sheet 2.3: Utilize pre-trained word embeddings (e.g., Word2Vec, GloVe) to create a heatmap representing the similarity between words. Choose a set of words and visualize their pairwise similarities. Task Sheet 2.4: Pick a set of words and plot their embeddings in a 2D or 3D scatter plot. Analyze the relationships and clusters among the words. Task Sheet 2.5: Build a Word2Vec model from scratch using a small text corpus. Train the model to generate word embeddings for words in the corpus. Task Sheet 2.6: Implement the training of a GloVe (Global Vectors for Word Representation) model using a given text dataset. Experiment with different hyperparameters. Task Sheet 2.7: Choose a pre-trained word embedding model (e.g., Word2Vec, GloVe, FastText) and use it to obtain word vectors for

	a set of words. Explore the semantic relationships. Task Sheet 2.8: Implement a function to calculate the similarity between two word vectors. Test it with words that should have similar and dissimilar meanings. Task Sheet 2.9: Use word embeddings to build a sentiment analysis model. Train the model to classify text sentiment (positive, negative, neutral) using a dataset with labeled sentiments.

Information Sheet 2: Analyze features

Learning Objective:

After completion of this information sheet, the learners will be able to explain, define and interpret the following contents:

2.1 Implement Feature extraction

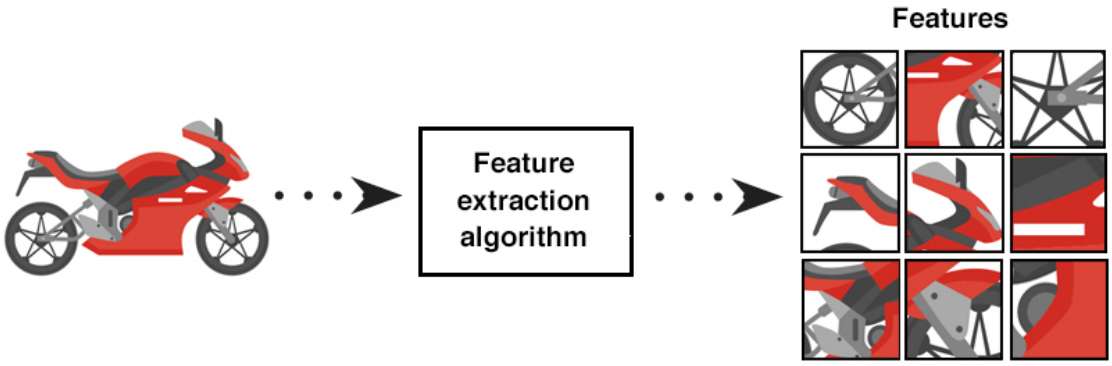
2.2 Visualization of word vectors are implemented with word cloud, histogram, heatmap, plots and tableau.

2.3 Exercise Embedding models are exercised

2.4 Implement Pre-trained word embedding

2.1. Implement Feature Extraction

Feature extraction is the process of transforming raw data into a format that is suitable for input into machine learning algorithms. It involves selecting relevant features and transforming them into a numerical representation that the machine learning model can process.



Example:

In NLP (Natural Language Processing), **word embeddings** are often used as features for text data. The most common embeddings are pre-trained word vectors that capture semantic relationships between words.

2.2. Visualization of Word Vectors

Visualization of word vectors allows for the representation of word similarities and relationships in a human-understandable form. Different techniques such as **Word Clouds**, **Histograms**, **Heatmaps**, **Plots**, and **Tableau** are used for this purpose.

1. Word Cloud

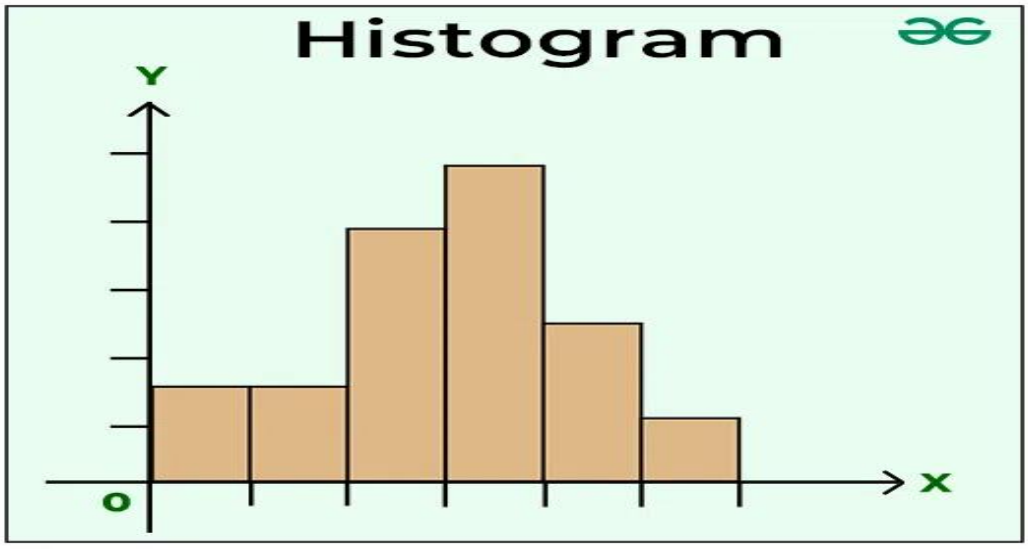
A **word cloud** is a popular way to visualize text data, where the size of each word indicates its frequency in a given corpus.

Example Code (Python):

```
from wordcloud import WordCloud
import matplotlib.pyplot as plt

text = "deep learning artificial intelligence neural networks machine learning"
wordcloud = WordCloud().generate(text)

plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```

2. Histogram

A **histogram** can be used to show the frequency distribution of word frequencies or word embeddings.

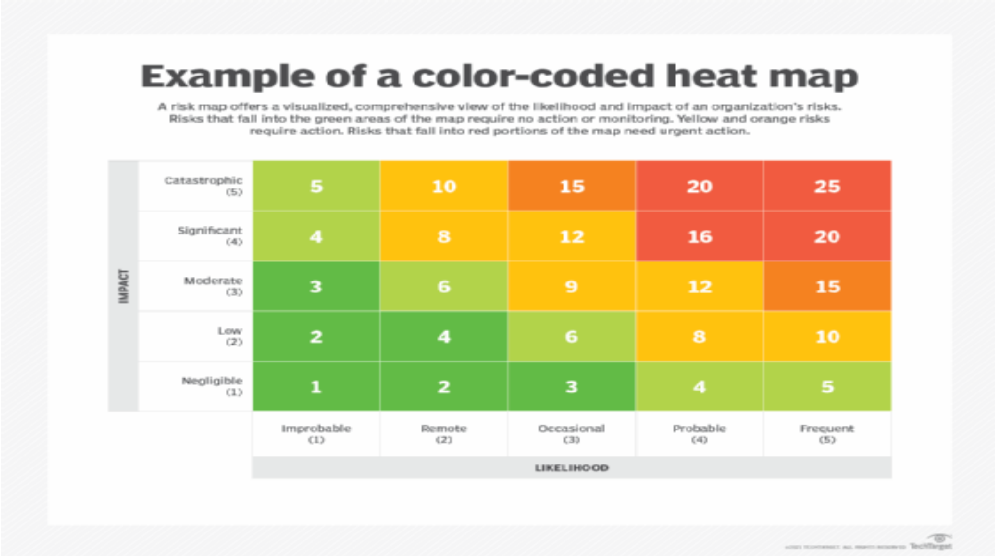


Example Code (Python):

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

# Example embedding values for words
word_embeddings = np.random.rand(100, 50)  # 100 words, 50 dimensions
embedding_sums = np.sum(word_embeddings, axis=1)

plt.hist(embedding_sums, bins=30)
plt.title("Histogram of Word Embedding Sums")
plt.xlabel("Sum of Embedding Values")
plt.ylabel("Frequency")
plt.show()
```

3. Heatmap

A **heatmap** can be used to visualize the relationships between different words (via cosine similarity between word vectors).



Example Code (Python):

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

```
# Example cosine similarity matrix
cosine_similarity_matrix = np.random.rand(10, 10)

sns.heatmap(cosine_similarity_matrix, annot=True, cmap="YlGnBu")
plt.title("Heatmap of Word Cosine Similarities")
plt.show()
```

4. Tableau

Tableau can be used for creating more interactive and detailed visualizations. You can export data (like word embeddings or cosine similarity matrices) to a CSV file and then use Tableau to create more complex visualizations, such as interactive scatter plots.

2.3. Exercise Embedding Models

Word Representation

Word representation refers to how words are encoded into numerical vectors. Pre-trained word embeddings like **Word2Vec**, **GloVe**, and **FastText** represent words as dense vectors in a high-dimensional space.

Embedding Matrix

An **embedding matrix** is a collection of word vectors, where each word in the vocabulary corresponds to a row in the matrix.

Example Code for Word Embedding Matrix:

```
import numpy as np

# Example vocabulary and word vectors
vocabulary = ['apple', 'banana', 'cherry']
embedding_matrix = np.random.rand(len(vocabulary), 50)  # 50-dimensional vectors

# Mapping each word to its vector in the matrix
word_to_idx = {word: idx for idx, word in enumerate(vocabulary)}
word_vector = embedding_matrix[word_to_idx['apple']]
print(word_vector)
```

Word2Vec

Word2Vec is a model that learns word representations by predicting context words given a target word (CBOW) or predicting a target word given context words (Skip-gram).

FastText

FastText is an extension of Word2Vec that considers subword information, representing words as bags of character n-grams.

GloVe (Global Vectors for Word Representation)

GloVe is another word embedding model that learns word vectors by factorizing the word co-occurrence matrix.

2.4. Implement Pre-trained Word Embedding

Implications of Pre-trained Word Vectors

Pre-trained word vectors are vectors that have been trained on large corpora like Wikipedia or the Common Crawl dataset. They can be fine-tuned for specific tasks or used as features for other models.

Example (Loading Pre-trained GloVe Vectors):

```
import numpy as np


# Load GloVe vectors
def load_glove_vectors(glove_file):
    word_to_vec_map = {}
    with open(glove_file, 'r', encoding='utf-8') as f:
        for line in f:
            values = line.split()
            word = values[0]
            word_to_vec_map[word] = np.asarray(values[1:], dtype='float32')
    return word_to_vec_map
```

```
# Example usage
glove_file = 'glove.6B.50d.txt'  # GloVe file with 50-dimensional vectors
word_to_vec = load_glove_vectors(glove_file)
vector_for_king = word_to_vec['king']
print(vector_for_king)
```

Tuning Word Vectors

Tuning pre-trained word vectors involves adjusting them based on a new dataset (e.g., through **fine-tuning** in transfer learning) to better suit the specific task.

Embedding Model Evaluation (Intrinsic and Extrinsic)

- **Intrinsic Evaluation**: Measures the quality of the embeddings directly (e.g., using tasks like word similarity or analogy).
- **Extrinsic Evaluation**: Measures the impact of embeddings on downstream tasks like classification or sentiment analysis.

Example of Intrinsic Evaluation:

```
from scipy.spatial.distance import cosine

# Example: Cosine similarity between two word vectors
similarity = 1 - cosine(word_to_vec['king'], word_to_vec['queen'])
print(f"Cosine Similarity: {similarity}")
```

Example of Extrinsic Evaluation:

After using pre-trained embeddings in a classification task, evaluate the model's performance on a test set (e.g., using accuracy, precision, or recall).

Self-Check Sheet 2: Analyze Features

Q1: How can word vectors be visualized with a word cloud?

Q2: What is the purpose of using a histogram in word vector visualization?

Q3: What is word representation in the context of embedding models?

Q4: What are Word2Vec, FastText, and GloVe?

Q5: What are the implications of using pre-trained word vectors?

Q6: How can word vectors be tuned?

Q7: How is embedding model evaluation conducted?

Answer Sheet 2: Analyze Features

Q1: How can word vectors be visualized with a word cloud?
Answer: A word cloud is a visual representation where the size of each word indicates its frequency or importance. In the context of word vectors, you can generate a word cloud to highlight words based on their similarity scores or relevance to certain topics, giving an overall sense of common or key words in a dataset.

Q2: What is the purpose of using a histogram in word vector visualization?
Answer: A histogram helps visualize the distribution of word vector values or similarity scores among words. For example, it can show how frequently certain word similarity scores appear, which can help understand the spread of words across different dimensions or categories.

Q3: What is word representation in the context of embedding models?
Answer: Word representation refers to encoding words as numerical vectors. These vectors capture semantic meaning by placing similar words close together in a multi-dimensional space, enabling models to understand relationships between words.

Q4: What are Word2Vec, FastText, and GloVe?
Answer:

- **Word2Vec**: A model developed by Google that uses context to learn word relationships. It creates embeddings based on neighboring words, focusing on either the Skip-gram or Continuous Bag of Words (CBOW) methods.
- **FastText**: An extension of Word2Vec developed by Facebook that represents words as subword units (n-grams), allowing it to handle out-of-vocabulary words by combining these subword units.
- **GloVe (Global Vectors)**: Developed by Stanford, GloVe captures word relationships based on global word co-occurrence in a corpus, resulting in embeddings that represent both word similarity and analogy relationships.

Q5: What are the implications of using pre-trained word vectors?
Answer: Pre-trained word vectors allow models to start with a strong understanding of language, reducing training time and improving performance, especially in scenarios with limited data. They can generalize well but may not fully capture domain-specific nuances without additional tuning.

Q6: How can word vectors be tuned?
Answer: Word vectors can be fine-tuned on specific tasks or domains by continuing to train them with task-specific data, allowing them to adjust to the unique linguistic patterns of the target dataset and improve task performance.

Q7: How is embedding model evaluation conducted?
Answer:

- **Intrinsic evaluation**: Measures the quality of embeddings based on semantic similarity or analogy tasks (e.g., “king - man + woman = queen”).
- **Extrinsic evaluation**: Assesses the performance of embeddings in downstream NLP tasks like sentiment analysis or machine translation. This approach is task-dependent and shows how well the embeddings improve model accuracy in practical applications.

Task Sheet 2.1: Use a dataset of text (e.g., news articles or tweets) and create a word cloud to visualize the frequency of words. Experiment with adjusting parameters like word frequency and font size.

Detail Steps for Task Implementation:

Step 1: Data Collection

1. **Choose a Dataset**:
    - Collect a dataset of text data, such as news articles, tweets, or any large collection of text. Ensure that the dataset is in a text format (e.g., CSV, JSON, or plain text files).
2. **Load the Dataset**:
    - Load the dataset into a programming environment using Python libraries such as pandas or json.

python
Copy

```
import pandas as pd

# Load dataset (e.g., CSV or JSON)
data = pd.read_csv("path_to_your_data.csv")
text_data = data['text_column']  # Assuming the column with text is called 'text_column'
```

Step 2: Text Preprocessing

1. **Text Cleaning**:
    - Remove unnecessary characters, URLs, stopwords, punctuation, and convert text to lowercase for better consistency in word frequency analysis.

python
Copy

```
import re
import nltk
from nltk.corpus import stopwords

# Download NLTK stopwords if not already installed
nltk.download('stopwords')

stop_words = set(stopwords.words('english'))

# Function to clean text
def clean_text(text):
    text = re.sub(r"http\S+", "", text)  # Remove URLs
    text = re.sub(r"[^a-zA-Z\s]", "", text)  # Remove punctuation and non-alphabetic characters
    text = text.lower()  # Convert to lowercase
    text = " ".join([word for word in text.split() if word not in stop_words])  # Remove stopwords
    return text
```

```
# Clean the text data
cleaned_data = text_data.apply(clean_text)
```

Step 3: Generate the Word Cloud

1. **Install Word Cloud Library**:
    - Use the wordcloud library to generate the word cloud.

bash
Copy
```
pip install wordcloud
```

2. **Create the Word Cloud**:
    - Generate the word cloud using the cleaned text data. You can adjust parameters like font size, maximum words, and frequency threshold.

python
Copy
```
from wordcloud import WordCloud
import matplotlib.pyplot as plt

# Join all the text into one string
text = " ".join(cleaned_data)

# Create and display the word cloud
wordcloud = WordCloud(width=800, height=400, max_words=100, background_color='white').generate(text)

# Display the word cloud
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")  # Turn off axis
plt.show()
```

Step 4: Experiment with Parameters

1. **Adjust Word Frequency**:
    - You can adjust the min_font_size, max_font_size, and max_words parameters to control the word cloud's appearance.

python
Copy
```
# Adjust font size and frequency
wordcloud = WordCloud(width=800, height=400, max_words=100, background_color='white',
                      min_font_size=10, max_font_size=100).generate(text)
```

2. **Change Colors and Layout**:
    - You can also change the background color, and experiment with different color palettes to enhance the visualization.

python
Copy
# Use a custom color palette
wordcloud = WordCloud(width=800, height=400, max_words=100, background_color='white',
colormap='coolwarm').generate(text)

Step 5: Save and Present the Results

1. **Save the Word Cloud**:
    - You can save the generated word cloud as an image for later use.

python
Copy
wordcloud.to_file("wordcloud_output.png")

2. **Analyze the Visualization**:
    - Look at the most frequent words in the word cloud and try to interpret their significance in the context of the dataset (e.g., are there any clear topics, trends, or issues?).

Task Sheet 2.2: Analyze the distribution of word frequencies in a given text corpus using a histogram. Identify the most common and least common words.

Detailed Steps:

1. **Setup Environment**:
    - Install required libraries such as TensorFlow/Keras, NumPy, Pandas, and Matplotlib (optional) for data manipulation, model building, and visualization.
2. **Dataset Selection and Preprocessing**:
    - Choose an appropriate dataset (e.g., **MNIST**, **CIFAR-10**, or a custom dataset).
    - Preprocess the data:
        - Normalize image data (scale pixel values to a range between 0 and 1).
        - For non-image data, apply standard normalization/standardization.
        - Split the dataset into **training**, **validation**, and **test** sets.
        - Use **Bootstrap Sampling** to create multiple subsets of the training data (sampling with replacement).
3. **Define the Base Neural Network Architecture**:
    - Select a simple neural network architecture (e.g., **Feedforward Neural Network** or **Convolutional Neural Network** if working with images).
        - For CNN: Input layer with appropriate image shape, followed by several convolutional and pooling layers, and a fully connected layer.
        - For Feedforward Neural Network: Input layer, hidden layers with **ReLU activation**, and output layer with **softmax** for classification or **linear activation** for regression.
4. **Bagging Implementation**:
    - **Bootstrap Sampling**: Create different subsets of the training data using bootstrap sampling (sampling with replacement).
    - **Train multiple instances**: Initialize the neural network model multiple times (e.g., 5 or 10 models).
        - Each model will be trained on a unique subset of the training data.
    - Train models independently and in parallel if possible, utilizing available computational resources.
5. **Train the Individual Neural Network Models**:
    - Train each instance of the neural network on its respective subset.
    - Monitor the training process with **early stopping** to prevent overfitting and ensure proper model convergence.
    - Track performance using appropriate metrics (e.g., accuracy for classification, MSE for regression).
6. **Prediction Aggregation**:
    - After training, use the ensemble of models to make predictions on the test data.
    - **For classification tasks**: Aggregate predictions using **majority voting** (each model casts a vote for a class, and the most frequent class is selected).

- **For regression tasks**: Aggregate predictions by averaging the outputs of all models.

7. **Evaluate the Bagging Ensemble**:
   - Evaluate the ensemble model's performance on the test set using suitable metrics like accuracy, precision, recall, F1 score (for classification), or mean squared error (for regression).
   - Compare the ensemble model's performance with individual models trained on the full dataset to assess the impact of bagging.
8. **Model Optimization**:
   - If needed, tune the base model's hyperparameters (e.g., number of layers, number of neurons, learning rate) to enhance performance.
   - Experiment with the number of models in the ensemble (e.g., 5 models vs. 10 models) to determine the optimal ensemble size.
9. **Document Findings and Visualize Results**:
   - Summarize the architecture of the neural network used for the ensemble and individual models.
   - Present performance comparison (e.g., accuracy, loss) in tables or graphs.
   - Visualize model performance through metrics such as accuracy curves, confusion matrices, or bar plots comparing individual models versus the ensemble.

Task Sheet 2.3: Utilize pre-trained word embeddings (e.g., Word2Vec, GloVe) to create a heatmap representing the similarity between words. Choose a set of words and visualize their pairwise similarities.

Detailed Steps

- **Step 1: Install and Import Required Libraries**
  - Install necessary libraries: gensim, matplotlib, seaborn.
  - Import libraries for word embedding processing and visualization.
- **Step 2: Load Pre-Trained Word Embeddings**
  - Use **Word2Vec** or **GloVe** pre-trained models.
  - Load embeddings with **Gensim** or other suitable libraries.
- **Step 3: Select Words for Similarity Analysis**
  - Choose a list of words (e.g., related to a specific domain like technology, nature, etc.).
- **Step 4: Obtain Word Vectors for the Selected Words**
  - Retrieve word vectors for each word from the pre-trained model.
  - Handle missing words in the model appropriately.
- **Step 5: Compute Cosine Similarities**
  - Calculate pairwise cosine similarity between word vectors.
  - Cosine similarity measures the angle between vectors, with values closer to 1 indicating high similarity.
- **Step 6: Visualize the Similarities Using a Heatmap**
  - Use **seaborn** and **matplotlib** to plot the cosine similarity matrix as a heatmap.
  - The heatmap provides a visual representation of word similarities.
- **Step 7: Interpret the Heatmap**
  - **Diagonal Values:** Always 1 (perfect similarity with itself).
  - **Off-Diagonal Values:** Represent cosine similarities between different words. Higher values indicate more similarity.

Task Sheet 2.4: Pick a set of words and plot their embeddings in a 2D or 3D scatter plot. Analyze the relationships and clusters among the words.

Detailed Steps:

1. **Word Selection:**
    - Choose a set of words that are related (e.g., animals, emotions, or food items).
    - Aim for a set of 10-20 words for clarity and simplicity.
2. **Word Embedding Model:**
    - Use a pre-trained word embedding model such as Word2Vec, GloVe, or FastText.
    - Load the model into your environment (Python libraries like gensim can be used to load pre-trained models).
3. **Obtain Word Embeddings:**
    - For each word in your selected set, obtain its corresponding embedding vector from the model.
    - These embeddings will be in a high-dimensional space (e.g., 100-dimensional or 300-dimensional vectors).
4. **Dimensionality Reduction:**
    - To visualize the embeddings, reduce the dimensionality from the high-dimensional space to 2D or 3D.
    - Use methods such as **Principal Component Analysis (PCA)** or **t-Distributed Stochastic Neighbor Embedding (t-SNE)** to project the embeddings to lower dimensions.
5. **Plot the Embeddings:**
    - Create a scatter plot for 2D or a 3D scatter plot if using 3D reduction.
    - Use libraries like matplotlib or plotly in Python for plotting.
6. **Label the Points:**
    - Label each point in the plot with the corresponding word for easy identification.
7. **Analyze the Plot:**
    - Observe the relationships between the words in the plot.
    - Look for clusters or groupings of similar words based on their meanings (e.g., animals might form a cluster).
    - Analyze if semantically similar words are placed closer to each other in the plot.
8. **Write Analysis:**
    - Provide an interpretation of the clusters or relationships. For example, “Words related to animals like ‘dog,’ ‘cat,’ and ‘elephant’ are grouped closely together, indicating that the model has captured their semantic similarity.”
    - Identify any outliers or unexpected groupings and consider why they may occur.

Task Sheet 2.5: Build a Word2Vec model from scratch using a small text corpus. Train the model to generate word embeddings for words in the corpus.

Detailed Steps:

1. **Corpus Preparation:**
    - Select a small text corpus (a few paragraphs or a short text dataset).
    - Preprocess the corpus by:
        - Converting all text to lowercase.
        - Removing punctuation, special characters, and stop words.
        - Tokenizing the text into words.
2. **Word2Vec Model Architecture:**
    - Understand that Word2Vec uses two primary models:
        - **CBOW (Continuous Bag of Words):** Predicts the target word based on its context.
        - **Skip-Gram:** Predicts context words based on the target word.
    - For simplicity, choose one model to implement (commonly, Skip-Gram is easier to understand).
3. **Preprocessing:**
    - Tokenize the corpus into a list of words (a list of lists where each sublist contains words from a sentence).
    - Create a vocabulary list that includes all unique words in the corpus.
    - Generate one-hot encoded vectors for words (or map each word to an integer).
4. **Model Implementation:**
    - Initialize:
        - **Input layer:** One-hot encoded vectors for context words.
        - **Hidden layer:** Randomly initialized weights (matrix) for mapping context to embeddings.
        - **Output layer:** One-hot encoded target word (for prediction).
    - The model consists of:
        - **Forward Pass:** Multiply input vector with weights to get the hidden layer, then apply activation (typically softmax).
        - **Backpropagation:** Adjust weights using gradient descent to minimize error.
5. **Training the Model:**
    - Use stochastic gradient descent (SGD) or another optimization technique.
    - Train the model over multiple epochs until the embeddings converge and show stable word relationships.
    - Use a loss function like **cross-entropy** to measure prediction accuracy.
6. **Extract Word Embeddings:**
    - After training, extract the learned word embeddings from the hidden layer.
    - These embeddings represent the words in a high-dimensional space and capture semantic relationships.

7. **Visualize Word Embeddings:**
    - Reduce the dimensionality of the embeddings (use PCA or t-SNE).
    - Visualize the embeddings in a 2D or 3D scatter plot (using matplotlib or plotly).
8. **Evaluate and Analyze:**
    - Analyze the word embeddings to identify semantic relationships.
    - Example: Check if similar words (e.g., synonyms) are clustered together.
9. **Iteration (Optional):**
    - If the results are not satisfactory, try adjusting parameters such as the size of the hidden layer, the learning rate, or the number of epochs.

Task Sheet 2.6: Implement the training of a GloVe (Global Vectors for Word Representation) model using a given text dataset. Experiment with different hyperparameters.

Detailed Steps:

1. **Dataset Preparation:**
    - Obtain or select a text dataset (can be a collection of documents, articles, or a small corpus).
    - Preprocess the text data by:
        - Converting all text to lowercase.
        - Removing punctuation, special characters, and stop words.
        - Tokenizing the text into words.
        - Creating a vocabulary of unique words in the dataset.
2. **Understand the GloVe Model:**
    - The GloVe model is based on factorizing the word co-occurrence matrix of the dataset.
    - It uses the following core idea: **The probability of two words occurring together is proportional to the product of their individual probabilities.**
    - The model computes word vectors by minimizing a cost function that measures the difference between predicted and actual word co-occurrence values.
3. **Create the Co-occurrence Matrix:**
    - Build a co-occurrence matrix where each entry represents the frequency of two words appearing together within a specific window in the text.
    - Choose a **window size** that defines how many words around a target word are considered as its context (typically between 5-10).
4. **Implement the GloVe Model:**
    - Initialize two matrices: **W** (for word vectors) and **C** (for co-occurrence counts).
    - Use the **weighted least squares** objective function to minimize the error between the predicted and observed co-occurrence counts.
    - The formula for the GloVe objective is: J=∑i,j=1Vf(Pij)(wiTwj+bi+bj−log⁡(Pij))2J = \sum_{i,j=1}^V f(P_{ij}) (w_i^T w_j + b_i + b_j - \log(P_{ij}))^2J=i,j=1∑Vf(Pij)(wiTwj+bi+bj−log(Pij))2 where:
        - wiw_iwi and wjw_jwj are the word vectors,
        - PijP_{ij}Pij is the co-occurrence probability,
        - bib_ibi and bjb_jbj are bias terms,
        - f(Pij)f(P_{ij})f(Pij) is a weighting function (often chosen as f(Pij)=(Pij)αf(P_{ij}) = (P_{ij})^\alphaf(Pij)=(Pij)α where α\alphaα is typically between 0.75 and 1).
5. **Experiment with Hyperparameters:**
    - **Embedding Dimensions**: Experiment with different values of embedding dimensions (e.g., 50, 100, 200, 300).

- **Window Size**: Vary the window size for the co-occurrence matrix (e.g., 5, 10, 15).
- **Learning Rate**: Test different learning rates (e.g., 0.05, 0.1, 0.2).
- **Regularization Parameter**: Experiment with regularization parameters to avoid overfitting (e.g., 0.001, 0.01).
- **Epochs**: Change the number of epochs for training to see the impact on convergence and quality of the embeddings.

6. **Training the Model:**
   - Train the GloVe model using stochastic gradient descent (SGD) or another optimization method.
   - Monitor the training progress by calculating the loss at regular intervals.
   - Save the resulting word vectors after training.
7. **Evaluate the Embeddings:**
   - Use various methods to evaluate the quality of the generated embeddings, such as:
     - **Cosine similarity** between word vectors to check for semantic similarity.
     - Visualize the embeddings in 2D or 3D using dimensionality reduction techniques like **t-SNE** or **PCA**.
     - Perform analogy tasks (e.g., "man" is to "king" as "woman" is to "queen").
8. **Parameter Tuning and Analysis:**
   - Adjust the hyperparameters based on the evaluation results. For example, if the embeddings are not capturing semantic relationships well, consider changing the embedding dimension or learning rate.
   - Perform a final evaluation with the best hyperparameters and analyze how well the model is capturing word relationships.

Task Sheet 2.7: Choose a pre-trained word embedding model (e.g., Word2Vec, GloVe, FastText) and use it to obtain word vectors for a set of words. Explore the semantic relationships.

Detailed Steps for Task Implementation:

1. **Select a Pre-Trained Model:**
    - Choose a popular pre-trained word embedding model:
        - **Word2Vec**: Trained on large corpora like Google News.
        - **GloVe**: Trained on datasets such as Wikipedia or Common Crawl.
        - **FastText**: Good for subword-level embeddings.
2. **Install Necessary Libraries:**
    - Install and import the required libraries:
        - **gensim** for Word2Vec and FastText.
        - **glove-python** for GloVe or **spaCy** for FastText integration.
        - **numpy, matplotlib, or plotly** for handling and visualizing embeddings.
3. **Load the Pre-Trained Model:**
    - Load the embedding model:
        - **Word2Vec**: Use gensim.models.KeyedVectors.load_word2vec_format().
        - **GloVe**: Use glove-python or **gensim**'s glove2word2vec() function.
        - **FastText**: Use gensim.models.fasttext.load_facebook_vectors().
4. **Select a Set of Words:**
    - Choose a set of words that represent different meanings, such as synonyms, antonyms, or related terms (e.g., "king," "queen," "man," "woman").
5. **Obtain Word Vectors:**
    - Retrieve the word vectors (embeddings) for the selected words from the pre-trained model.
6. **Explore Semantic Relationships:**
    - Use **cosine similarity** to measure similarity between word vectors.
        - Cosine similarity ranges from -1 (completely opposite) to 1 (completely similar), with 0 indicating no similarity.
    - Explore analogies (e.g., "king" is to "queen" as "man" is to "woman").
7. **Visualize Word Embeddings:**
    - Use dimensionality reduction techniques like **PCA** or **t-SNE** to reduce high-dimensional embeddings to 2D or 3D.
    - Visualize the relationships between words on a scatter plot.
8. **Analyze the Results:**
    - Based on the visualized scatter plot and cosine similarity results, analyze the relationships:
        - Words closer together represent similar meanings.
        - Words farther apart represent unrelated meanings.
    - Use analogy tasks to assess how well the model captures semantic relationships (e.g., gender or functional roles).
9. **Experiment with Other Models (Optional):**

- Experiment with different models (e.g., **Word2Vec**, **FastText**, **GloVe**) and compare the results. Models may capture relationships differently depending on training data and methodology.

Task Sheet 2.8: Implement a function to calculate the similarity between two word vectors. Test it with words that should have similar and dissimilar meanings.

Detailed Steps

1. **Import Required Libraries:**
    - Ensure the necessary libraries are installed:
        - **gensim**: For working with pre-trained word embeddings.
        - **numpy**: For performing vector operations.
2. **Load Pre-trained Word Embeddings:**
    - Select and load a pre-trained word embedding model such as **Word2Vec**, **GloVe**, or **FastText**.
    - Choose the model based on your preference or task requirements (e.g., **Word2Vec** is often used for general tasks like similarity analysis).
3. **Implement Similarity Function:**
    - Implement a function to calculate **cosine similarity** between two word vectors. Cosine similarity is used to measure how similar two vectors are by comparing the angle between them.
        - **Cosine Similarity Formula:** Cosine Similarity=A·B‖A‖‖B‖\text{Cosine Similarity} = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}
        - Where A\mathbf{A} and B\mathbf{B} are the word vectors, and ·\cdot represents the dot product. The norm (or magnitude) of each vector is represented by ‖A‖\|\mathbf{A}\| and ‖B‖\|\mathbf{B}\|.
4. **Test the Similarity Function:**
    - Choose a set of words that have known semantic relationships. For instance:
        - Words that should be **similar**: "king" and "queen" (gender-related words).
        - Words that should be **dissimilar**: "king" and "apple" (unrelated terms).
    - Test the cosine similarity function using these word pairs to check the output similarity values.
5. **Interpret the Results:**
    - The cosine similarity value between semantically related words (e.g., "king" and "queen") should be high (close to 1).
    - The cosine similarity between unrelated words (e.g., "king" and "apple") should be low (close to 0).
6. **Analyze Edge Cases:**
    - Experiment with other word pairs that may test the limits of the model, such as:
        - Related pairs: "dog" and "cat" (related animals).
        - Opposites: "hot" and "cold" (antonyms).
    - Use these edge cases to analyze how well the model captures various semantic relationships like synonyms, antonyms, and other related concepts.

Task Sheet 2.9: Use word embeddings to build a sentiment analysis model. Train the model to classify text sentiment (positive, negative, neutral) using a dataset with labeled sentiments.

Detailed Steps for Sentiment Analysis Implementation:

1. **Dataset Preparation:**
    - **Obtain a sentiment-labeled dataset** suitable for training the model. Popular options include:
        - IMDb Reviews: Movie review sentiment analysis.
        - Sentiment140: A large Twitter dataset labeled with sentiment (positive, negative, neutral).
        - Amazon Reviews: Product reviews labeled with sentiment.
    - **Preprocess the text data:**
        - Clean the text by removing unnecessary punctuation, special characters, and stop words.
        - Tokenize the text into individual words.
        - Label the data into categories such as positive, negative, and neutral based on the sentiment of the text.
2. **Word Embeddings:**
    - **Use pre-trained word embeddings** such as Word2Vec, GloVe, or FastText to convert text into word vectors.
        - You can use pre-trained embeddings from libraries like **gensim** or **spaCy**.
        - Alternatively, you can load pre-trained embeddings like **Google's Word2Vec** or **GloVe** embeddings.
3. **Text Vectorization:**
    - **Convert words into vectors** by averaging the word embeddings for all words in a sentence or document.
        - The idea is to get a fixed-size vector for each text by averaging the embeddings of the individual words.
4. **Split Dataset:**
    - **Split the dataset** into a training set and a testing set (e.g., 80% for training and 20% for testing).
        - Use libraries like **scikit-learn** to handle the dataset splitting.
5. **Build Sentiment Analysis Model:**
    - Use **machine learning models** such as Logistic Regression, Random Forest, or SVM for classification, or deep learning-based models like **LSTM** or **CNN**.
        - Train the model on the word vectors from the training set and evaluate it on the test set.
6. **Model Evaluation:**

- **Evaluate the model** using classification metrics like accuracy, precision, recall, and F1-score to assess its performance on the test set.

7. **Hyperparameter Tuning (Optional):**
   - **Experiment with different models and hyperparameters**, as well as switching between different word embedding types (e.g., from Word2Vec to GloVe or FastText).
     - Use techniques like **GridSearchCV** or **RandomizedSearchCV** to find the best hyperparameters for your model.
8. **Testing the Model:**
   - After training and tuning the model, **test it on unseen data** to assess how well it performs with new inputs.
     - Input a sample sentence or review and predict its sentiment using the trained model.

By following these steps, you'll build a sentiment analysis model that can accurately classify text into different sentiment categories based on the embeddings and machine learning model used.

Learning Outcome 3: Implement DL features

Assessment Criteria	1. ANN and CNN are comprehended 2. CNN Variations are implemented 3. Optimization of hyperparameters is implemented 4. Recurrent neural networks are implemented 5. Ensemble of DL Models is implemented
Condition and Resource	1. Actual workplace or training environment 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil and Eraser 7. Internet Facilities 8. Whiteboard and Marker 9. Imaging Device (Digital camera, scanner etc.)
Content	▪ ANN and CNN ▪ Network design ▪ Convolution operation ▪ Max-pooling operation ▪ Building network ▪ Training, testing and validation ▪ CNN Variations ▪ AlexNet ▪ VGG-19 ▪ GoogLeNet ▪ ResNet-18 ▪ ResNet-152 ▪ MobileNet ▪ Recurrent neural networks ▪ Why RNNs ▪ GRU, LSTM ▪ Forwardpropagation ▪ Backpropagation ▪ Vanishing gradient ▪ Exploding gradient ▪ Bidirectional RNNs ▪ Optimization of hyperparameters ▪ Understanding parameters and hyperparameters ▪ Tuning hyperparameters ▪ Effect of hyperparameter tuning ▪ Ensemble of DL Models ▪ Why Ensemble ▪ How to Ensemble ▪ Average Ensemble ▪ Weighted Ensemble ▪ Voting Ensemble

Activity/ task/ Job	• Implement a basic CNN for image classification using a popular dataset (e.g., CIFAR-10). Experiment with different architectures, such as varying the number of layers and filter sizes. • Fine-tune a pre-trained CNN model (e.g., VGG16, ResNet) on a specific dataset for a different image classification task. Evaluate its performance on the new task. • Extend a CNN architecture with batch normalization layers. Train the model on an image classification task and compare its performance with a baseline CNN. • Implement a CNN with Spatial Pyramid Pooling to handle input images of different sizes. Evaluate its performance on an image classification task. • Implement grid search to tune hyperparameters (e.g., learning rate, batch size) for a deep learning model. Use a validation set to find the best combination. • Apply random search for hyperparameter tuning on a deep learning model. Compare the results with grid search and analyze the advantages. • Use a Bayesian optimization library (e.g., Optuna) to optimize hyperparameters for a deep learning model. Compare the results with grid search and random search. • Implement a simple RNN to generate text character by character. Train the model on a text corpus and generate new sequences. • Build an LSTM-based model for sentiment analysis on a dataset with labeled text reviews. Explore the use of word embeddings. • Implement a bidirectional LSTM model for Named Entity Recognition. Train the model on a dataset with labeled entities (e.g., names, locations). • Implement an ensemble model that combines the predictions of a CNN and an LSTM model. Use it for a task such as video classification. • Create a stacked ensemble by combining predictions from diverse architectures, such as CNN, LSTM, and a simple feedforward neural network. Implement a bagging ensemble with multiple instances of the same neural network architecture. Train each instance on a different subset of the data.
Training Technique	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice

	5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Methods of Assessment	1. Written Test 2. Demonstration 3. Oral Questioning

Learning Experience 3: Implement DL features

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Implement DL features'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Implement DL features"	2. Read Information sheet 3: Implement DL features 3. Answer Self-check 3: Implement DL features 4. Check your answer with Answer key 3: Implement DL features
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job/Task Sheet and Specification Sheet Task Sheet 3.1: Implement a voting ensemble for classification Task Sheet 3.2: Build an average ensemble of two CNN models. Task Sheet 3.3 : Implement a simple LSTM layer for text classification. Job Sheet 3: Developing an Ensemble Deep Learning Model for Video Sentiment Analysis

Information Sheet 3: Implement DL features

Learning Objective:

After completion of this information sheet, the learners will be able to explain, define and interpret the following contents:

3.1 ANN and CNN

3.2 Implement CNN Variations

3.3 Implement Optimization of hyperparameters

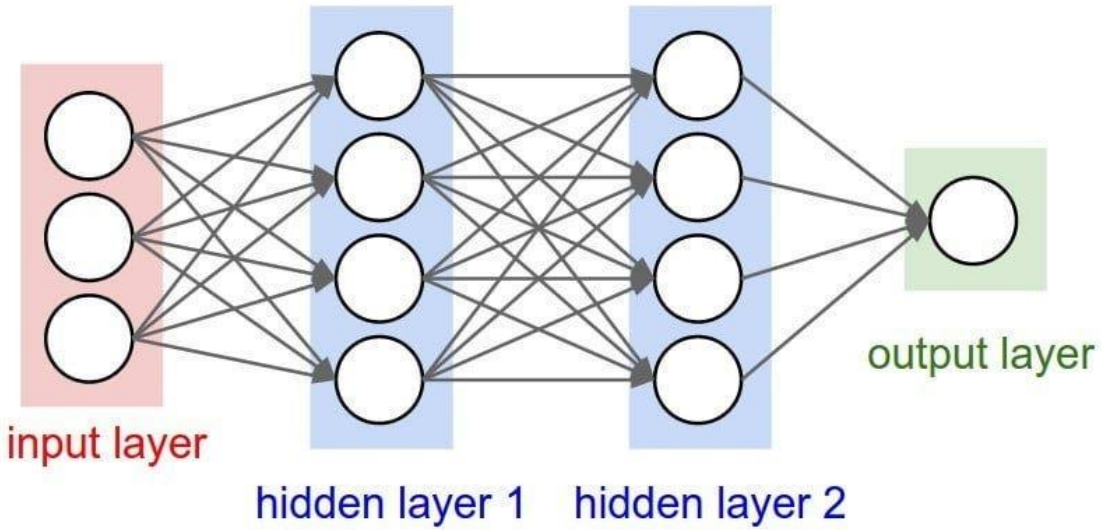
3.4 Implement Recurrent neural networks

3.5 Implement Ensemble of DL Models

3.1. ANN and CNN

Artificial Neural Networks (ANN)

Definition: Artificial Neural Networks (ANN) are computational models inspired by the way biological neural networks in the brain process information. They consist of layers of interconnected neurons where each connection has an associated weight that adjusts during training to minimize the loss function.



Key Concepts:

- **Input Layer:** Accepts input data (e.g., images, text).
- **Hidden Layers:** Perform transformations on the input data and learn features.

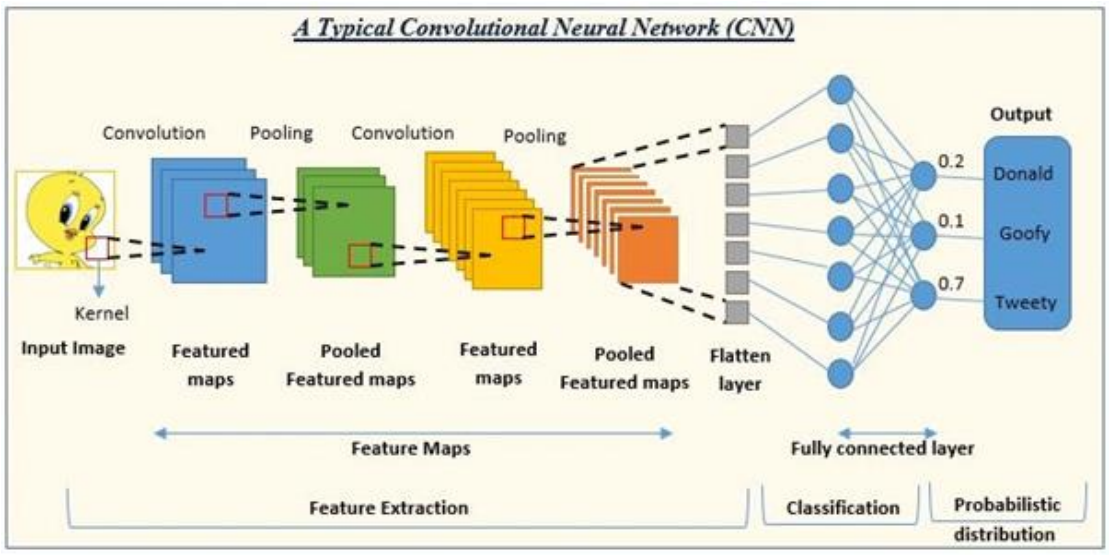
- **Output Layer:** Produces the result (e.g., class prediction).

Example:
Building a simple neural network for image classification using Keras.

```
from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense


model = Sequential([

    Dense(128, activation='relu', input_shape=(784,)),  # Input layer with 784 features (MNIST images)

    Dense(10, activation='softmax')  # Output layer for 10 classes (digits 0-9)

])


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

model.summary()
```

Convolutional Neural Networks (CNN)

Definition: CNNs are a class of deep neural networks commonly used for analyzing visual data. CNNs consist of convolution layers that automatically extract spatial hierarchies from images, followed by pooling layers that reduce dimensionality.



Key Concepts:

- **Convolution Operation:** This operation applies a filter (kernel) to an image to extract features (e.g., edges, textures).
- **Max-Pooling Operation:** This reduces the spatial dimensions (height, width)

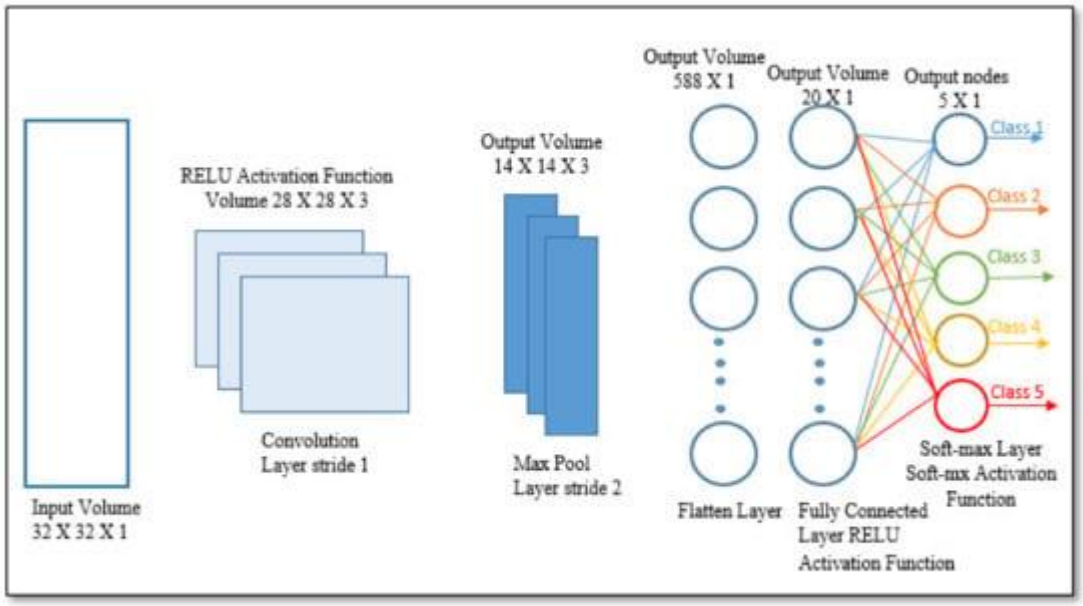
by taking the maximum value in a given region, reducing the computation for the next layer.

Building a Simple CNN

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),  # Convolutional
layer
    MaxPooling2D((2, 2)),  # Max pooling layer
    Flatten(),  # Flatten the 2D data into 1D
    Dense(128, activation='relu'),  # Fully connected layer
    Dense(10, activation='softmax')  # Output layer for 10 classes (digits)
])

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

3.2. Implement CNN Variations

CNNs have evolved into more complex architectures over time. Here are a few common variations:



1. **AlexNet:**
    - **Architecture:** A deep CNN with 5 convolutional layers and 3 fully connected layers.
    - **Contribution:** Revolutionized image classification by winning the 2012 ImageNet competition with a large margin.
2. **VGG-19:**
    - **Architecture:** Consists of 19 layers (16 convolutional layers and 3 fully connected layers).
    - **Contribution:** Demonstrated that deep architectures improve performance.
3. **GoogLeNet (Inception):**
    - **Architecture:** Introduces Inception blocks, which use multiple filter sizes at each layer to capture multi-scale features.
    - **Contribution:** Efficient architecture with a reduced number of parameters.
4. **ResNet-18 and ResNet-152:**
    - **Architecture:** Uses residual blocks to solve the vanishing gradient problem in very deep networks. ResNet-18 has 18 layers, and ResNet-152 has 152 layers.
    - **Contribution:** Introduced skip connections that allow gradients to flow more effectively.
5. **MobileNet:**
    - **Architecture:** Designed for mobile devices. Uses depthwise separable convolutions to reduce the number of parameters and computation.
    - **Contribution:** Makes CNNs more efficient for real-time applications on mobile devices.

3.3. Implement Optimization of Hyperparameters

Understanding Parameters vs. Hyperparameters:

- **Parameters:** These are learned from the training data (e.g., weights and biases).
- **Hyperparameters:** These are set prior to training and control the learning process (e.g., learning rate, batch size, number of epochs).

Tuning Hyperparameters:

The process of selecting the optimal hyperparameters can significantly improve model performance. Common methods include:

- **Grid Search:** Exhaustively search through a predefined set of hyperparameter values.
- **Random Search:** Randomly sample from the hyperparameter space.
- **Bayesian Optimization:** Uses probabilistic models to find the most optimal values based on prior evaluations.

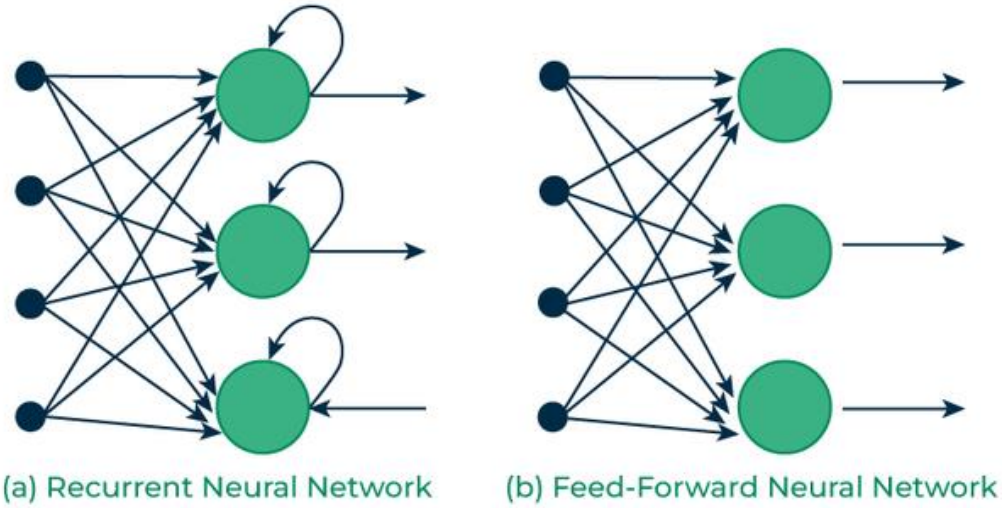
Example: Hyperparameter Tuning with Keras Tuner:

```
import keras_tuner as kt

def build_model(hp):
    model = Sequential([
        Dense(hp.Int('units', min_value=32, max_value=512, step=32), activation='relu', input_shape=(784,)),
        Dense(10, activation='softmax')
    ])

    model.compile(optimizer=hp.Choice('optimizer', values=['adam', 'sgd']), loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model

tuner = kt.Hyperband(build_model, objective='val_accuracy', max_epochs=10, directory='my_dir')
tuner.search(train_images, train_labels, epochs=10, validation_data=(test_images, test_labels))
```

3.4. Implement Recurrent Neural Networks (RNN)

Why RNNs?

RNNs are used for sequence data (e.g., text, time series, audio) because they have connections that loop back, allowing them to retain information from previous time steps.



Types of RNNs:

- **Simple RNNs:** The basic RNN architecture, but suffers from vanishing gradients.
- **GRU (Gated Recurrent Units):** An improvement over simple RNNs, using gating mechanisms to control information flow.
- **LSTM (Long Short-Term Memory):** A type of RNN that overcomes the vanishing gradient problem with special units for remembering long-term dependencies.

Example (LSTM for Text Classification):

```
from tensorflow.keras.models import Sequential

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


model = Sequential([

   Embedding(input_dim=10000, output_dim=128, input_length=200),  # Word embedding layer

   LSTM(64),  # LSTM layer

   Dense(10, activation='softmax')  # Output layer

])


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

model.summary()
```

Vanishing and Exploding Gradients:

- **Vanishing Gradient:** Gradients become very small, preventing effective training of deep networks.
- **Exploding Gradient:** Gradients become too large, causing instability during training.

Both issues are mitigated by LSTM and GRU cells.

3.5. Implement Ensemble of DL Models

Why Ensemble?

Ensemble methods combine the predictions of multiple models to improve performance by reducing variance, bias, or both.

Types of Ensemble Methods:

1. **Average Ensemble:** The predictions of multiple models are averaged to make the final decision.
2. **Weighted Ensemble:** Similar to average ensemble, but different models have different weights in the final decision, reflecting their importance.
3. **Voting Ensemble:** Each model in the ensemble votes for a class, and the class with the most votes is chosen as the final prediction.

Example (Voting Ensemble):

```
from sklearn.ensemble import VotingClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier

# Define base models
model1 = SVC(kernel='linear', probability=True)
model2 = KNeighborsClassifier()

# Create a VotingClassifier
ensemble_model = VotingClassifier(estimators=[('svm', model1), ('knn', model2)], voting='soft')

# Train the ensemble model
ensemble_model.fit(X_train, y_train)
```

Self-Check Sheet 3: Implement DL Features

Q1: What is network design in ANN and CNN?

Q2: What is the convolution operation in CNNs?

Q3: What is max-pooling, and why is it used in CNNs?

Q4: What are training, testing, and validation in CNNs?

Q5: What is AlexNet, and why is it significant?

Q6: How does VGG-19 differ from AlexNet?

Q7: What is ResNet, and why is it important?

Q8: Why are RNNs used?

Q9: What are GRUs and LSTMs in RNNs?

Q10: What is backpropagation in RNNs?

Answer Sheet 3: Implement DL Features

Q1: What is network design in ANN and CNN?
Answer: Network design refers to the architecture of a neural network, including the number and types of layers, the number of neurons in each layer, and the connections between them. In CNNs, this design often includes convolutional layers, pooling layers, and fully connected layers, each tailored for processing spatial or temporal data.

Q2: What is the convolution operation in CNNs?
Answer: The convolution operation is a mathematical operation where a filter (kernel) slides across an input image, detecting features like edges or textures by focusing on spatial hierarchies. This process extracts important patterns from images while reducing the need for manual feature engineering.

Q3: What is max-pooling, and why is it used in CNNs?
Answer: Max-pooling is a down-sampling operation in CNNs that selects the maximum value from a region of the feature map, reducing its size and emphasizing the most prominent features. This helps reduce computational cost and prevents overfitting.

Q4: What are training, testing, and validation in CNNs?
Answer:

- **Training**: The model learns patterns by adjusting weights based on labeled data.
- **Validation**: A separate dataset is used during training to tune hyperparameters and assess model performance, helping to avoid overfitting.
- **Testing**: A final assessment on a separate dataset to evaluate the model's accuracy and generalizability.

Q5: What is AlexNet, and why is it significant?
Answer: AlexNet is a CNN architecture that popularized deep learning in image recognition by winning the ImageNet competition in 2012. It consists of multiple convolutional and pooling layers, utilizing ReLU activations and dropout to reduce overfitting.

Q6: How does VGG-19 differ from AlexNet?
Answer: VGG-19 is a deeper CNN architecture with 19 layers, known for its simplicity and use of small 3x3 filters throughout. It emphasizes depth in feature extraction, making it more accurate but computationally heavier than AlexNet.

Q7: What is ResNet, and why is it important?
Answer: ResNet (Residual Network) introduced skip connections (or residual connections)

to help mitigate the vanishing gradient problem in deep networks. ResNet-18 and ResNet-152 are different versions with 18 and 152 layers, respectively, allowing the training of extremely deep networks with high accuracy.

Q8: Why are RNNs used?
Answer: RNNs are designed to handle sequential data, such as time series and language, by maintaining information across time steps. This enables them to capture temporal dependencies and patterns in sequences.

Q9: What are GRUs and LSTMs in RNNs?
Answer:

- **GRU (Gated Recurrent Unit)**: A type of RNN cell that simplifies computation by using fewer gates than LSTM while still mitigating the vanishing gradient problem.
- **LSTM (Long Short-Term Memory)**: An RNN cell with three gates (input, forget, output) that manages long-term dependencies more effectively, solving the vanishing gradient issue in traditional RNNs.

Q10: What is backpropagation in RNNs?
Answer: Backpropagation in RNNs, specifically Backpropagation Through Time (BPTT), involves calculating gradients across all time steps and updating weights, allowing the model to learn temporal dependencies.

Task Sheet 3.1: Implement a basic CNN for image classification using a popular dataset (e.g., CIFAR-10). Experiment with different architectures, such as varying the number of layers and filter sizes.

Steps to Implement:

1. **Set up the Development Environment:**
    - Install Python 3.x along with libraries such as TensorFlow, Keras, and Matplotlib.
    - Set up a **virtual environment** for managing dependencies (recommended).
    - Install the required libraries using a package manager like pip.
2. **Import Required Libraries:**
    - Import the necessary libraries for data loading, model building, and visualization.
3. **Load CIFAR-10 Dataset:**
    - Load the CIFAR-10 dataset using a built-in dataset function from the deep learning library you are using (e.g., TensorFlow).
4. **Pre-process the Data:**
    - Normalize the images so that the pixel values are in the range [0, 1].
    - Convert the labels to **categorical** format using one-hot encoding to prepare for classification.
5. **Define the CNN Architecture:**
    - Start with a basic architecture containing 2 convolutional layers.
    - Experiment by adding or removing layers, changing the number of filters in each layer, and adjusting the kernel sizes to see their effects on performance.
6. **Compile the Model:**
    - Choose an optimizer (e.g., **Adam**), loss function (e.g., **categorical cross-entropy**), and evaluation metric (e.g., **accuracy**) for compiling the model.
7. **Train the Model:**
    - Train the model on the training dataset for a set number of epochs.
    - Use a validation set during training to monitor the model's performance.
8. **Evaluate the Model:**
    - After training, evaluate the model's performance using the test dataset to determine its final accuracy.
9. **Experiment with Different Architectures:**
    - Change aspects of the model such as the number of convolutional layers, the number of filters per layer, and the size of the filters. Test these changes to see how they affect the model's performance.
10. **Visualize the Results:**
    - Plot the training and validation accuracy to visually analyze the model's performance over epochs and detect signs of overfitting.

11. **Save and Load the Model (Optional):**
    - Save the trained model to disk so that you can load it later for inference or further training without needing to retrain it from scratch.

Task Sheet 3.2: Fine-tune a pre-trained CNN model (e.g., VGG16, ResNet) on a specific dataset for a different image classification task. Evaluate its performance on the new task.

Steps to Implement:

1. **Set up the Development Environment:**
    - Install Python 3.x along with essential libraries like TensorFlow, Keras, and Matplotlib.
    - It's recommended to set up a virtual environment to manage dependencies.
    - Install the required libraries using a package manager like pip.
2. **Import Required Libraries:**
    - Import the necessary libraries for model building, data loading, and evaluation.
3. **Load the Pre-trained Model:**
    - Select a pre-trained model such as VGG16 or ResNet50. These models are already trained on large datasets like ImageNet and can be adapted for other tasks.
4. **Load and Pre-process Your Dataset:**
    - Choose the dataset you want to work with (e.g., flowers, animals, or a custom dataset).
    - Pre-process the images to ensure they match the input format of the pre-trained model (e.g., resizing images to a specific size, normalizing pixel values).
    - Set up data generators to load the images and augment the training dataset if needed.
5. **Freeze the Pre-trained Layers:**
    - Freeze the layers of the pre-trained model to retain the learned features from the initial training and avoid altering them during fine-tuning.
6. **Add Custom Classifier Layers:**
    - Add custom layers on top of the pre-trained model to adapt it for your new classification task (e.g., adding a fully connected layer with a softmax activation for the final classification).
7. **Compile the Model:**
    - Compile the model by selecting an appropriate optimizer, loss function, and evaluation metric for your classification task.
8. **Train the Model:**
    - Train the model using your dataset, initially keeping the pre-trained layers frozen, so only the new layers you added are trained.
9. **Unfreeze Some Pre-trained Layers (Optional):**
    - After the initial training, you may choose to unfreeze some of the pre-trained layers and fine-tune them for improved performance.
10. **Evaluate the Model:**
    - Evaluate the model's performance on the test dataset to assess how well it generalizes to unseen data.

11. **Visualize the Results:**
    - Plot the training and validation accuracy/loss over epochs to visualize the model's performance and check for overfitting.
12. **Save the Model (Optional):**
    - Once the model has been fine-tuned, save it to disk for later use or deployment.

Task Sheet 3.3: Extend a CNN architecture with batch normalization layers. Train the model on an image classification task and compare its performance with a baseline CNN.

Steps to Implement:

1. **Set up the Development Environment:**
    - Install Python 3.x and essential libraries like TensorFlow, Keras, and Matplotlib.
    - It is recommended to set up a virtual environment to manage dependencies effectively.
    - Install the necessary dependencies using a package manager like pip.
2. **Import Required Libraries:**
    - Import the libraries required for building, training, and evaluating the models.
3. **Load and Pre-process the Dataset:**
    - Choose a dataset for image classification (e.g., CIFAR-10, MNIST, or a custom dataset).
    - Normalize the images by scaling the pixel values to the range [0, 1] for consistent input to the neural network.
4. **Define the Baseline CNN Architecture:**
    - Define a basic Convolutional Neural Network (CNN) architecture without batch normalization. This will serve as the baseline for comparison with the extended model.
5. **Define the Extended CNN Architecture with Batch Normalization:**
    - Modify the CNN architecture by adding BatchNormalization layers after the convolutional layers. This will help to stabilize and speed up the training process.
6. **Train the Baseline CNN Model:**
    - Train the baseline CNN model on the chosen dataset and save the training history for later evaluation.
7. **Train the Extended CNN Model with Batch Normalization:**
    - Train the extended CNN model, which includes the batch normalization layers, on the same dataset and save the training history.
8. **Evaluate Both Models:**
    - Evaluate both the baseline model and the extended model with batch normalization on the test dataset to compare their performance.
9. **Compare Performance:**
    - Visualize the training and validation accuracy/loss curves for both models to compare how they perform over epochs. This helps in assessing the impact of batch normalization on model performance.
10. **Draw Conclusions:**
    - Analyze the test accuracy and training curves to understand the impact of batch normalization. Typically, batch normalization helps to stabilize and accelerate the training process, improving accuracy and reducing overfitting.

11. **Save the Models (Optional):**
    - Optionally, save both models for future use, such as further training or deployment.

Task Sheet 3.4: Implement a CNN with Spatial Pyramid Pooling to handle input images of different sizes. Evaluate its performance on an image classification task.

Steps to Implement:

1. **Set up the Development Environment:**
    - Install Python 3.x along with essential libraries like TensorFlow, Keras, and Matplotlib.
    - It's recommended to set up a virtual environment to manage dependencies effectively.
    - Use pip to install the required dependencies.
2. **Import Required Libraries:**
    - Import the necessary libraries for building, training, and evaluating the models.
3. **Define the Spatial Pyramid Pooling (SPP) Layer:**
    - Implement the Spatial Pyramid Pooling (SPP) layer, which helps in handling input images of varying sizes by applying pooling at multiple spatial scales.
4. **Load and Pre-process the Dataset:**
    - Choose a suitable dataset for image classification (e.g., CIFAR-10, MNIST, or your custom dataset).
    - Pre-process the images by resizing or normalizing them to prepare them for the model.
5. **Define the CNN Architecture with Spatial Pyramid Pooling:**
    - Define a CNN architecture that incorporates the Spatial Pyramid Pooling layer to handle images of different sizes without resizing them to a fixed size.
6. **Train the Model:**
    - Train the model on the image dataset. As the model can handle images of varying sizes, there is no need to resize the images to a fixed size.
7. **Evaluate the Model:**
    - After training, evaluate the model’s performance on the test dataset to check its accuracy.
8. **Visualize the Results:**
    - Plot the training and validation accuracy/loss curves to assess the model's performance over epochs.
9. **Compare Performance with a Baseline CNN:**
    - To evaluate the effectiveness of the Spatial Pyramid Pooling layer, compare the performance of the SPP-based model with a baseline CNN without the SPP layer.
10. **Analyze the Results:**
    - Analyze the results by comparing the test accuracy, loss, and training curves of both the SPP-based model and the baseline CNN to determine the benefits of incorporating SPP.

11. **Save the Models (Optional):**
    - Optionally, save both models for future use or deployment to avoid retraining them.

Task Sheet 3.5: Implement grid search to tune hyperparameters (e.g., learning rate, batch size) for a deep learning model. Use a validation set to find the best combination.

Detailed Steps

1. **Dataset Preparation:**
    - **Obtain a dataset** suitable for training a deep learning model (e.g., image classification, text classification).
    - **Preprocess the dataset** (e.g., tokenization for text, resizing and normalization for images).
    - Split the dataset into **training**, **validation**, and **test sets**.
2. **Model Definition:**
    - Choose a **deep learning architecture** (e.g., CNN, LSTM, etc.) depending on the task.
    - Implement the model using libraries like **TensorFlow** or **Keras**.
3. **Define Hyperparameter Grid:**
    - Define a grid of hyperparameters to tune. For example:
        - **Learning rate**: [0.001, 0.01, 0.1]
        - **Batch size**: [32, 64, 128]
        - **Number of epochs**: [10, 20, 30]
4. **Grid Search Implementation:**
    - Use **GridSearchCV** or a custom grid search loop to evaluate different combinations of hyperparameters.
    - For each combination of hyperparameters, train the model using the **training set** and evaluate it on the **validation set**.
5. **Model Evaluation:**
    - For each hyperparameter combination, record the performance (e.g., accuracy, loss, or other relevant metrics) on the **validation set**.
    - After completing the grid search, select the best-performing combination of hyperparameters.
6. **Train the Final Model:**
    - Once the best hyperparameters are found, train the model again on the **entire training set** using those optimal hyperparameters.
    - Evaluate the model on the **test set** to assess its performance on unseen data.
7. **Hyperparameter Tuning with Early Stopping (Optional):**
    - Implement **early stopping** to prevent overfitting during training, which can improve the generalization of the model.

Task Sheet 3.6: Apply random search for hyperparameter tuning on a deep learning model. Compare the results with grid search and analyze the advantages.

Detailed Steps:

1. **Prepare Environment**:
    - Install necessary libraries (e.g., TensorFlow, Keras, Scikit-learn, NumPy, Matplotlib).
    - Ensure all necessary tools and equipment are available.
2. **Dataset Selection**:
    - Choose a dataset suitable for deep learning (e.g., MNIST, CIFAR-10).
    - Split the dataset into training, validation, and test sets.
3. **Deep Learning Model Setup**:
    - Design a simple neural network or a predefined model architecture using a deep learning library like TensorFlow/Keras.
    - Define the model's layers, activation functions, optimizer, and loss function.
4. **Hyperparameter Selection**:
    - Identify a set of hyperparameters to tune, such as:
        - Learning rate
        - Batch size
        - Number of epochs
        - Number of hidden layers and neurons
        - Activation function
        - Optimizer (SGD, Adam, etc.)
5. **Random Search Hyperparameter Tuning**:
    - Use Scikit-learn's RandomizedSearchCV or any other random search library to perform random search on the hyperparameters.
    - Define a hyperparameter space (parameter grid) for each hyperparameter to be searched.
6. **Grid Search Hyperparameter Tuning**:
    - Implement grid search using Scikit-learn's GridSearchCV or any other grid search implementation.
    - Use the same hyperparameter space as used in random search.
7. **Model Evaluation**:
    - For each method (random search and grid search), evaluate the model on the validation set using performance metrics (e.g., accuracy, loss).
    - Track the best combination of hyperparameters for each method.
8. **Comparison and Analysis**:
    - Compare the performance of random search and grid search based on the evaluation metrics.
    - Analyze the computational cost (time taken) and accuracy/performance of both methods.

- Discuss the advantages of random search over grid search and vice versa.

9. **Document the Findings**:
    - Provide a comprehensive analysis of the results.
    - Summarize key insights and suggestions for which method may be better under different scenarios.

Task Sheet 3.7: Use a Bayesian optimization library (e.g., Optuna) to optimize hyperparameters for a deep learning model. Compare the results with grid search and random search.

Detailed Steps:

1. **Prepare Environment**:
    - Install necessary libraries:
        - Optuna for Bayesian optimization.
        - TensorFlow/Keras for building the deep learning model.
        - Scikit-learn for grid search and random search.
    - Ensure proper version compatibility.
2. **Dataset Selection**:
    - Choose a dataset suitable for deep learning (e.g., MNIST, CIFAR-10).
    - Split the dataset into training, validation, and test sets.
3. **Deep Learning Model Setup**:
    - Design a neural network architecture using TensorFlow/Keras.
    - Define the layers, activation functions, optimizer, loss function, and evaluation metric.
4. **Hyperparameter Selection**:
    - Identify key hyperparameters for optimization, such as:
        - Learning rate
        - Batch size
        - Number of epochs
        - Number of hidden layers and neurons
        - Dropout rate
        - Optimizer choice (e.g., Adam, SGD)
5. **Implement Optuna for Bayesian Optimization**:
    - Define an objective function that accepts hyperparameters and returns the model's validation performance.
    - Use Optuna's study.optimize() to perform hyperparameter search.
    - Set the search space for each hyperparameter (e.g., loguniform for learning rate, int for batch size).
6. **Implement Grid Search for Comparison**:
    - Use Scikit-learn's GridSearchCV to perform grid search on the same hyperparameters.
    - Define a grid of hyperparameters and evaluate the model using cross-validation.
7. **Implement Random Search for Comparison**:
    - Use Scikit-learn's RandomizedSearchCV to perform random search on the same hyperparameters.
    - Define the parameter distributions for random search.
8. **Model Evaluation**:

- Train and evaluate the model for each search method (Bayesian, grid, and random) using the validation set.
- Track the performance metrics (e.g., accuracy, loss) for each method.

9. **Comparison and Analysis**:
   - Compare the results of Bayesian optimization, grid search, and random search based on:
     - Performance metrics (accuracy, loss).
     - Computational cost (time taken).
     - Hyperparameter combinations found.
   - Discuss the advantages and disadvantages of Bayesian optimization compared to grid and random search.
10. **Document the Findings**:
    - Provide a detailed analysis of the results.
    - Summarize key insights on which method is most suitable under different circumstances.
    - Highlight the computational efficiency of Bayesian optimization.

Task Sheet 3.8: Implement a simple RNN to generate text character by character. Train the model on a text corpus and generate new sequences.

Detailed Steps:

1. **Prepare the Environment**:
    - Install necessary libraries:
        - TensorFlow/Keras for building the RNN.
        - NumPy for data manipulation.
        - Matplotlib (optional) for visualizations.
    - Set up the programming environment (e.g., Jupyter Notebook, Python IDE).
2. **Load and Preprocess the Text Corpus**:
    - Choose a text corpus (e.g., a novel, lyrics, or a collection of articles).
    - Load the text and preprocess it:
        - Convert all text to lowercase (optional).
        - Remove any non-alphabetic characters (optional).
        - Create a mapping from characters to integers and vice versa.
3. **Prepare the Data for the RNN**:
    - Create sequences of characters from the text corpus.
        - Define a fixed sequence length (e.g., 100 characters).
        - For each sequence, predict the next character.
    - Split the data into input sequences (X) and target sequences (y).
4. **Build the Simple RNN Model**:
    - Design an RNN architecture using Keras:
        - Input layer (embedding layer if needed).
        - Recurrent layer (use SimpleRNN, GRU, or LSTM).
        - Output layer with softmax activation for character prediction.
    - Compile the model using an appropriate optimizer (e.g., Adam) and loss function (e.g., sparse categorical crossentropy).
5. **Train the Model**:
    - Train the RNN on the character sequences.
        - Use the input sequences (X) and target sequences (y).
        - Define the number of epochs and batch size.
    - Monitor the training process to ensure convergence and adjust hyperparameters if necessary.
6. **Text Generation**:
    - Once the model is trained, use it to generate new text:
        - Provide an initial "seed" text (a starting sequence of characters).
        - Generate the next character by predicting the next value and appending it to the seed.
        - Repeat this process to generate a sequence of characters.
7. **Evaluate the Model**:
    - Analyze the generated text in terms of coherence, creativity, and relevance.

- Fine-tune the model if needed by adjusting parameters such as sequence length, learning rate, or model architecture.

8. **Document the Findings**:
    - Provide a report on the implementation process.
    - Include an analysis of the generated text and discuss potential improvements.
    - Suggest use cases for the generated text model (e.g., chatbot, creative writing assistant).

Task Sheet 3.9: Build an LSTM-based model for sentiment analysis on a dataset with labeled text reviews. Explore the use of word embeddings.

Detailed Steps:

1. **Prepare the Environment**:
    - Install necessary libraries:
        - **TensorFlow/Keras** for building and training the LSTM model.
        - **NumPy** for numerical operations.
        - **Pandas** for data manipulation.
        - **Matplotlib** for visualization (optional).
        - **NLTK** or **SpaCy** for text preprocessing (optional).
    - Ensure that the programming environment (e.g., Jupyter Notebook, Python IDE) is ready.
2. **Load and Preprocess the Dataset**:
    - Choose a sentiment analysis dataset (e.g., IMDB movie reviews, Amazon product reviews).
    - **Load the dataset** using Pandas or another library.
        - Make sure the dataset has text reviews and corresponding sentiment labels (e.g., positive/negative).
    - **Preprocess the text data**:
        - Convert all text to lowercase.
        - Remove stop words, punctuation, and special characters (optional).
        - Tokenize the text (split the text into words).
        - Optionally, **lemmatize** or **stem** words to reduce inflectional forms.
    - **Split the dataset** into training, validation, and test sets (e.g., 80/10/10 split).
3. **Explore Word Embeddings**:
    - Download pre-trained **word embeddings** like **GloVe** or **Word2Vec**.
        - GloVe embeddings can be downloaded from the official GloVe website (https://nlp.stanford.edu/projects/glove/).
        - Load the word embeddings into memory and create a word-to-index dictionary.
    - **Map words in the dataset** to their corresponding embedding vectors. If a word isn't in the embedding vocabulary, you can either:
        - Use a random vector.
        - Use a zero vector.
    - Alternatively, you can **train your own word embeddings** using the Embedding layer in Keras.
4. **Build the LSTM Model**:
    - Define an **LSTM-based architecture** using Keras:
        - **Input layer**: Tokenized and padded sequences of text.
        - **Embedding layer**: Use pre-trained word embeddings (if available), otherwise initialize randomly.

        - **LSTM layer(s)**: One or more LSTM layers to capture sequential patterns in the reviews.
        - **Dense layer**: A fully connected layer with 128 or 64 units.
        - **Output layer**: A dense layer with a single unit and **sigmoid activation** for binary classification (positive/negative).
    - **Compile the model** using an appropriate optimizer (e.g., Adam) and loss function (e.g., binary cross-entropy).
5. **Train the Model**:
    - Train the model on the training data using the following parameters:
        - Batch size (e.g., 32 or 64).
        - Number of epochs (e.g., 10-20).
        - Use validation data to monitor overfitting.
        - Optionally, use callbacks like **EarlyStopping** to prevent overfitting.
6. **Evaluate the Model**:
    - After training, evaluate the model on the test set.
        - Calculate the accuracy, precision, recall, and F1 score for sentiment classification.
        - Optionally, plot **confusion matrix** and **ROC curve** for evaluation.
7. **Generate Predictions**:
    - Use the trained model to generate predictions for unseen text reviews.
    - Display or print some of the predicted results alongside the actual sentiment labels for inspection.
8. **Analyze Results**:
    - Examine the model's performance in terms of accuracy, loss, and other metrics.
    - If the model is underperforming, consider:
        - Tuning hyperparameters.
        - Adding more LSTM layers.
        - Using a different word embedding model.
        - Data augmentation or further data preprocessing.
9. **Document the Findings**:
    - Summarize the steps taken, model architecture, and results.
    - Include visualizations, such as training/validation loss curves or confusion matrices.
    - Discuss potential improvements, such as using bidirectional LSTM layers or fine-tuning hyperparameters.

Task Sheet 3.10: Implement a bidirectional LSTM model for Named Entity Recognition. Train the model on a dataset with labeled entities (e.g., names, locations).

Detailed Steps:

1. **Prepare the Environment**:
    - Install necessary libraries:
        - **TensorFlow/Keras** for building the Bidirectional LSTM model.
        - **NumPy** for numerical operations.
        - **Pandas** for handling datasets.
        - **NLTK** or **SpaCy** for preprocessing (optional).
    - Set up a programming environment (e.g., Jupyter Notebook, Python IDE).
2. **Load and Preprocess the Dataset**:
    - Choose a labeled dataset suitable for Named Entity Recognition, such as:
        - **CoNLL-03** dataset (with entities labeled as PER, LOC, ORG, etc.).
        - **OntoNotes 5** or any other labeled NER dataset.
    - **Load the dataset** into a pandas DataFrame (or equivalent).
    - **Preprocess the text data**:
        - Tokenize the text into words.
        - Split the text into sentences or sequences of words.
        - Label each word with its corresponding entity (or 'O' for non-entity words).
3. **Prepare Data for NER**:
    - **Create a word-to-index and label-to-index mapping**:
        - Create a dictionary mapping each word to a unique integer.
        - Create a dictionary for the entity labels (e.g., PER, LOC, ORG, O).
    - **Pad the sequences**:
        - Use padding to ensure all input sequences are of the same length, especially for batching.
    - **Split the data** into training, validation, and test sets (e.g., 80/10/10 split).
4. **Implement the Bidirectional LSTM Model**:
    - Define an architecture for the Bidirectional LSTM model using Keras:
        - **Input layer**: Tokenized and padded sequences of words.
        - **Embedding layer**: Convert words into embeddings (use pretrained word embeddings like **GloVe** or **Word2Vec**).
        - **Bidirectional LSTM layer**: Apply the Bidirectional LSTM to capture dependencies in both directions (forward and backward).
        - **Dropout layer** (optional): To prevent overfitting.
        - **Dense layer**: A fully connected layer to predict the entity label for each word.
        - **Output layer**: A softmax activation layer to output probabilities for each entity label (including 'O' for non-entity).

- **Compile the model** using an appropriate optimizer (e.g., Adam) and loss function (e.g., categorical cross-entropy).

5. **Train the Model**:
   - Train the Bidirectional LSTM model on the training data:
     - Use the input sequences (X) and corresponding entity labels (Y).
     - Set the number of epochs (e.g., 10-20) and batch size (e.g., 32).
     - Monitor the training and validation loss to ensure the model is learning without overfitting.
   - Optionally, use **callbacks** like **EarlyStopping** to prevent overfitting.
6. **Evaluate the Model**:
   - After training, evaluate the model on the test set:
     - Compute evaluation metrics such as accuracy, precision, recall, and F1-score for each entity type (e.g., PER, LOC, ORG).
     - Use a confusion matrix to analyze the performance on different entity classes.
7. **Entity Recognition and Prediction**:
   - Once the model is trained and evaluated, use it to predict entities on new text:
     - Input a sentence or text, and use the trained model to predict entity labels.
     - Extract and display the recognized entities along with their labels (e.g., "John" as PER, "New York" as LOC).
8. **Improve Model (Optional)**:
   - If the model is underperforming, consider the following improvements:
     - Fine-tune the hyperparameters (e.g., number of LSTM units, learning rate).
     - Use additional layers like CRF (Conditional Random Fields) for sequence prediction.
     - Add more training data or use data augmentation techniques.
9. **Document the Findings**:
   - Provide a detailed analysis of the model's performance, including its strengths and weaknesses.
   - Include visualizations such as training/validation loss curves and confusion matrices.
   - Suggest areas for future work, such as experimenting with different architectures (e.g., Transformer-based models).

Task Sheet 3.11: Implement an ensemble model that combines the predictions of a CNN and an LSTM model. Use it for a task such as video classification.

Detailed Steps:

1. **Prepare the Environment**:
    - Install necessary libraries:
        - **TensorFlow/Keras** for building the CNN, LSTM, and ensemble models.
        - **NumPy** for data manipulation.
        - **OpenCV** or **PIL** for reading video frames.
        - **Matplotlib** for visualizations (optional).
    - Set up the programming environment (e.g., Jupyter Notebook, Python IDE).
2. **Dataset Preparation**:
    - Choose a suitable **video classification dataset**, such as:
        - **UCF101** or **Kinetics-400** (large-scale video datasets with labeled classes).
        - Preprocess the video data:
            - Extract frames from each video (e.g., using OpenCV).
            - Resize each frame to a consistent size (e.g., 224x224 pixels for CNN).
            - Normalize pixel values (e.g., scaling between 0 and 1).
            - Split the dataset into training, validation, and test sets.
    - Optionally, **augment the data** (e.g., using random rotations, flips, or color jitter) to improve model generalization.
3. **Build the CNN Model**:
    - Create a **CNN model** to extract spatial features from individual frames:
        - Input layer: Each frame resized to 224x224 pixels (or another suitable size).
        - Several convolutional layers with **ReLU activation**.
        - Max-pooling layers to reduce dimensionality.
        - Fully connected layer(s) for final classification.
        - Output layer: A softmax activation to output class probabilities.
    - Compile the CNN model using an appropriate optimizer (e.g., Adam) and loss function (e.g., categorical cross-entropy).
4. **Build the LSTM Model**:
    - Create an **LSTM model** to capture temporal dependencies across video frames:
        - Input layer: Sequences of frames (e.g., 30 frames per video).
        - Convert frames into a sequence format (e.g., (batch_size, sequence_length, height, width, channels)).
        - Flatten each frame's output using a **Flatten** or **GlobalAveragePooling** layer.

        - Pass the sequence of features into an **LSTM layer** to capture the temporal relationship.
        - Dense layer(s) for final classification.
        - Output layer: A softmax activation to output class probabilities.
    - Compile the LSTM model using an appropriate optimizer (e.g., Adam) and loss function (e.g., categorical cross-entropy).
5. **Ensemble Model Design**:
    - The ensemble model will combine the outputs of the CNN and LSTM models.
        - **Model 1 (CNN)**: Use the CNN model to extract spatial features from the video frames.
        - **Model 2 (LSTM)**: Use the LSTM model to capture temporal dependencies across frames.
    - **Fusion Strategy**: Combine the predictions of the CNN and LSTM models:
        - One possible approach is to use a **concatenation** layer to merge the outputs of both models.
        - Another approach is to **average** or **weight** the predictions of the CNN and LSTM models.
    - The combined output will be passed through a final **Dense layer** to get the final classification prediction.
6. **Train the Ensemble Model**:
    - Train the ensemble model using the video data:
        - Use both CNN and LSTM components for feature extraction.
        - Use appropriate loss functions and optimizers for training.
        - Monitor training and validation accuracy to ensure convergence.
        - Optionally, use **callbacks** like **EarlyStopping** to prevent overfitting.
7. **Evaluate the Model**:
    - After training, evaluate the performance of the ensemble model on the test set:
        - Measure classification accuracy, precision, recall, and F1 score.
        - Optionally, plot confusion matrices and ROC curves to evaluate performance across classes.
8. **Predict on New Data**:
    - Use the trained ensemble model to classify new unseen videos:
        - Extract frames from the new video.
        - Pass the frames through both CNN and LSTM models.
        - Combine their predictions and output the final class.
9. **Model Optimization and Improvement**:
    - If the model performs suboptimally, consider:
        - Tuning hyperparameters such as learning rate, number of LSTM units, or CNN architecture.
        - Experiment with different fusion strategies (e.g., weighted averaging of CNN and LSTM predictions).
        - Add more data or use transfer learning by fine-tuning pre-trained CNN models (e.g., ResNet, VGG16).

10. **Document the Findings**:

- Summarize the implementation, model architecture, and performance.
- Include any performance evaluations such as accuracy, precision, recall, or F1 score.
- Visualize key results, such as confusion matrices or classification reports.
- Discuss the potential applications of the model, such as video categorization, surveillance, or activity recognition.

Task Sheet 3.12: Create a stacked ensemble by combining predictions from diverse architectures, such as CNN, LSTM, and a simple feedforward neural network.

Detailed Steps:

1. **Prepare the Environment**:
    - Install the necessary libraries:
        - **TensorFlow/Keras** for building the CNN, LSTM, and Feedforward Neural Network (FNN).
        - **NumPy** for numerical computations.
        - **Pandas** for data handling.
        - **Matplotlib** for data visualization (optional).
    - Ensure the programming environment (e.g., Jupyter Notebook or Python IDE) is ready for coding.
2. **Dataset Selection and Preprocessing**:
    - Select a dataset suitable for the task (e.g., image classification, time-series prediction, or any other task).
        - If image classification: Use datasets like **CIFAR-10**, **MNIST**, or **ImageNet**.
        - If time-series prediction: Use datasets like **UCI Time Series Dataset** or any labeled time-series data.
    - **Preprocess the data**:
        - For images: Resize the images, normalize pixel values to the range [0, 1], and split the dataset into training, validation, and test sets.
        - For time-series: Normalize the data, split into sequences (if necessary), and divide into training, validation, and test sets.
3. **Model 1: Convolutional Neural Network (CNN)**:
    - Build a **CNN model** for spatial feature extraction:
        - Input layer: Process image input of size (height, width, channels) for image data (or suitable size for time-series).
        - Several convolutional layers with **ReLU activation** and **MaxPooling** layers.
        - Flatten the output from convolutional layers to pass to a dense layer.
        - Fully connected layer(s) and **softmax** output for classification (or regression if applicable).
    - Compile the CNN model with an appropriate optimizer (e.g., Adam) and loss function (e.g., categorical cross-entropy for classification).
4. **Model 2: Long Short-Term Memory Network (LSTM)**:
    - Build an **LSTM model** for capturing temporal patterns (especially useful for time-series or sequential data):
        - Input layer: Sequence data (e.g., time-steps for time-series or sequence of frames for video).
        - LSTM layer(s) to capture temporal dependencies in the data.

        - Optionally, apply **Dropout** layers to prevent overfitting.
        - Dense layer(s) followed by a **softmax** activation for classification (or other suitable output layer).
    - Compile the LSTM model with the optimizer and loss function suited for the task.
5. **Model 3: Feedforward Neural Network (FNN)**:
    - Build a **simple feedforward neural network (FNN)**:
        - Input layer: Flatten the input if it's not already in a 1D vector (e.g., for image data, flatten after CNN processing).
        - Several hidden dense layers with **ReLU activation**.
        - Output layer: Use a **softmax** activation for classification tasks or a linear activation for regression tasks.
    - Compile the FNN model using an appropriate optimizer (e.g., Adam) and loss function.
6. **Train the Individual Models**:
    - Train the CNN, LSTM, and FNN models separately on the dataset.
        - Use the training set for fitting the models.
        - Use validation data to monitor the performance and prevent overfitting (use **EarlyStopping** callback if necessary).
    - Track the accuracy or loss of each model and ensure that each model is properly trained.
7. **Stacking the Ensemble**:
    - After training the individual models (CNN, LSTM, FNN), combine their predictions to create the ensemble:
        - **Level-1 models**: The CNN, LSTM, and FNN outputs are passed as input to a second-level model (e.g., a simple **Logistic Regression**, **Dense Neural Network**, or another classifier).
        - **Stacking strategy**: The predictions from each model (CNN, LSTM, FNN) are concatenated (or averaged) and fed into the level-1 model to make the final prediction.
    - Use the predictions from the CNN, LSTM, and FNN as input features for the stacked model (i.e., the second-level model). You may also include additional features if necessary.
8. **Train the Stacked Model**:
    - Train the stacked model (level-1) on the predictions from the individual models (CNN, LSTM, FNN).
        - Use the training data (or a validation set) to fine-tune the stacked model.
        - You can use a simple **Logistic Regression** or a **Dense Neural Network** to combine the predictions.
        - Ensure that the stacking model is properly trained by monitoring its performance.
9. **Evaluate the Ensemble Model**:
    - After training the stacked ensemble model, evaluate its performance on the test set:

        - Compute metrics such as accuracy, precision, recall, F1 score, or other relevant evaluation metrics depending on the task.
        - Compare the performance of the ensemble model with the individual models (CNN, LSTM, FNN) to assess improvement.

10. **Generate Predictions**:

- Use the trained ensemble model to generate predictions on new, unseen data.
    - For each input, the CNN, LSTM, and FNN models will provide individual predictions.
    - The stacked model will combine these predictions to output the final result.

11. **Document the Findings**:

- Summarize the models used, the architecture of each model, and the ensemble method.
- Include performance metrics (accuracy, precision, recall, etc.) for the individual models and the ensemble model.
- Visualize the results, such as training curves, confusion matrix, or ROC curve, to provide insights into model performance.
- Discuss the effectiveness of the ensemble and potential improvements for future work.

Task Sheet 3.13: Implement a bagging ensemble with multiple instances of the same neural network architecture. Train each instance on a different subset of the data.

Detailed Steps:

1. **Prepare the Environment**:
    - Install the required libraries:
        - **TensorFlow/Keras** for creating neural network models.
        - **NumPy** for numerical computations.
        - **Pandas** for handling data and creating subsets.
        - **Matplotlib** for visualizations (optional).
    - Set up the programming environment (e.g., Jupyter Notebook or Python IDE).
2. **Dataset Selection and Preprocessing**:
    - Choose an appropriate dataset (e.g., image classification task like **CIFAR-10**, **MNIST**, or a custom dataset for regression or classification).
    - **Preprocess the dataset**:
        - Normalize the data (scale values to the range [0, 1] for image data, or standardize for other types of data).
        - Split the dataset into training, validation, and test sets (use **train_test_split** from sklearn).
    - For each subset, ensure that the data is shuffled and split into multiple random samples (bootstrap sampling).
3. **Define the Base Neural Network Model**:
    - Choose a simple neural network architecture (e.g., **Feedforward Neural Network** or **Convolutional Neural Network** for images).
        - Input layer: Ensure the correct shape for input data (e.g., for images, a size of (height, width, channels)).
        - Add hidden layers with **ReLU activation** and **Dropout** to avoid overfitting.
        - Final output layer: Softmax for classification tasks or a linear activation for regression tasks.
    - Compile the model using an appropriate optimizer (e.g., Adam) and loss function (e.g., categorical cross-entropy for classification tasks, mean squared error for regression).
4. **Bagging Implementation**:
    - **Bootstrap Sampling**: For each model in the ensemble, create a different subset of the data using bootstrap sampling (sampling with replacement).
        - Divide the dataset into multiple subsets (e.g., 5 or 10 subsets).
    - **Train Multiple Instances of the Neural Network**:
        - For each subset of the data, train an independent instance of the neural network model.

        - Use parallel processing if the computational resources allow for it (this will speed up training).
    - Each model will learn from a different subset of the training data, ensuring that each model has a unique perspective on the data.
5. **Train the Bagging Ensemble**:
    - Train each of the models using its respective subset of the training data.
    - Use a **training loop** to handle the training of each individual model, ensuring that each model's training is independent.
    - Optionally, use **early stopping** or monitor the training process to prevent overfitting and ensure that the models converge.
6. **Aggregate Predictions from the Ensemble**:
    - After training the models, use the ensemble to make predictions on new data (either from the validation or test set).
        - **For Classification Tasks**: Aggregate predictions by **voting** (majority vote for each class) across all models.
        - **For Regression Tasks**: Aggregate predictions by **averaging** the outputs of all models.
    - Ensure that each model's prediction is weighted equally in the aggregation process.
7. **Evaluate the Ensemble Model**:
    - Evaluate the performance of the ensemble model on the validation or test set:
        - Compute accuracy, precision, recall, F1 score (for classification tasks), or mean squared error (for regression tasks).
        - Compare the performance of the ensemble model against individual models trained on the full dataset to assess the effectiveness of the bagging approach.
    - **Cross-validation** can be used to further validate the model's performance and robustness.
8. **Optimize the Model**:
    - If necessary, tune the hyperparameters of the base neural network model (e.g., number of hidden layers, number of units in each layer, learning rate) to improve performance.
    - Experiment with the number of models in the ensemble to see if increasing or decreasing the number improves the performance.
9. **Document and Visualize Results**:
    - Summarize the architecture of the base neural network and the bagging ensemble.
    - Visualize the accuracy/loss curves during training for the individual models and the ensemble model.
    - Compare the performance metrics (accuracy, precision, recall, etc.) of the ensemble model and individual models.

Learning Outcome 4: Perform project development activities

Assessment Criteria	1. Projects are developed using DL 2. Transformer based models are interpreted 3. Specific models are designed 4. Evaluation of DL models are performed
Condition and Resource	1. Actual workplace or training environment 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil and Eraser 7. Internet Facilities 8. Whiteboard and Marker 9. Imaging Device (Digital camera, scanner etc.)
Content	▪ Projects are developed using DL o Handwritten character recognition o Image classification o Object recognition o Face detection o Recommendation system o Sentiment analysis o Emotion analysis o Text classification o Aggressive text detection o Multimodal meme detection ▪ Transformer based models are interpreted o Transformer based model definition o Why use Transformer based models o Transformer Vs. DL Models ▪ Specific models are designed o M-BERT o Distil-BERT o XML-R o ROBERTa ▪ Evaluation of DL models are performed o Performance matrices o Error Analysis
Activity/ task/ Job	• Develop a deep learning model for image classification using a convolutional neural network (CNN). Use a dataset of your choice and evaluate the model's performance.
Training Technique	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving

	8. Brainstorming
Methods of Assessment	1. Written Test 2. Demonstration 3. Oral Questioning

Learning Experience 4: Perform project development activities

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Exercise Deep Learning (DL)'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Perform project development activities"	2. Read Information sheet 3: Perform project development activities 3. Answer Self-check 3: Perform project development activities 4. Check your answer with Answer key 3: Perform project development activities
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job/Task Sheet and Specification Sheet Task Sheet 4.1: Develop a deep learning model for image classification using a convolutional neural network (CNN). Use a dataset of your choice and evaluate the model's performance.

Information Sheet4: Perform project development activities

Learning Objective:

After completion of this information sheet, the learners will be able to explain, define and interpret the following contents:

4.1 Projects are developed using DL

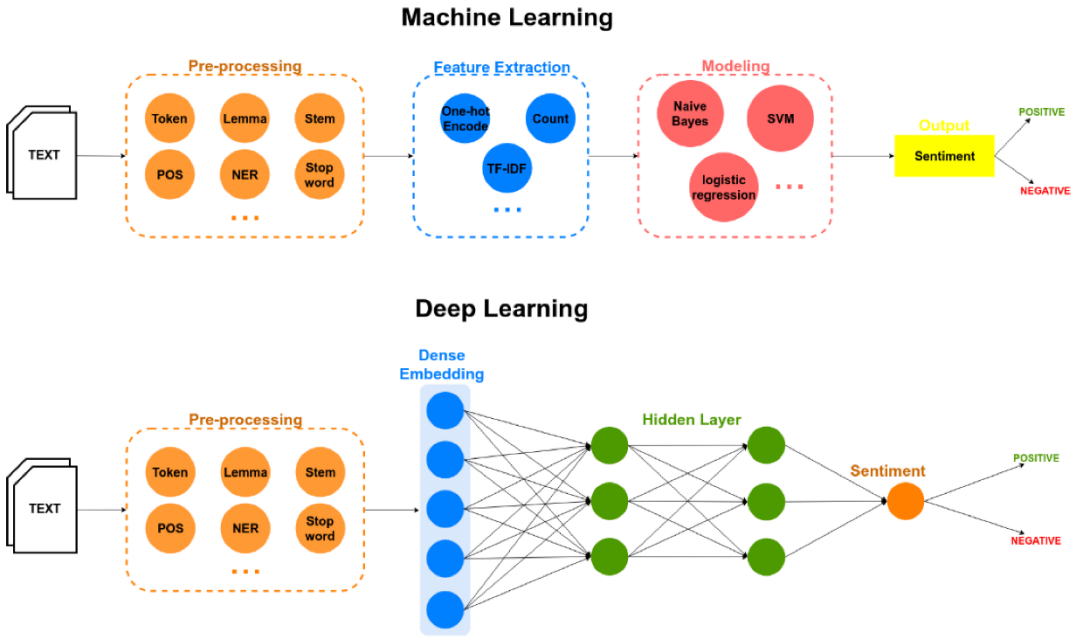
4.2 Interpret transformer-based models

4.3 Design Specific models

4.4 Perform Evaluation of DL models

4.1. Projects Developed Using Deep Learning (DL)

Common DL Projects:



Handwritten Character Recognition:

Example: Using CNN for digit classification on the MNIST dataset.

Code Example:

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, MaxPooling2D
from tensorflow.keras.utils import to_categorical
```

```
# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train.reshape(-1, 28, 28, 1).astype('float32') / 255
X_test = X_test.reshape(-1, 28, 28, 1).astype('float32') / 255
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Build CNN model
model = Sequential([
    Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=5, batch_size=128, validation_data=(X_test, y_test))
```

Image Classification:

Example: Classifying images using CNNs for categories such as "cat" and "dog."

Code Example: (Similar to Handwritten Recognition above, but applied to a dataset like CIFAR-10)

Object Recognition:

Example: Using models like YOLO or Faster R-CNN for object detection.

Tools: OpenCV, TensorFlow, PyTorch

Face Detection:

Example: Using Haar cascades or CNNs for detecting faces in images and videos.

Code Example: (Using OpenCV with pre-trained classifiers)

Recommendation System:

Example: Using deep learning for collaborative filtering or content-based recommendations.

Code Example: Building a neural network-based recommendation system.

Sentiment Analysis:

Example: Analyzing the sentiment of a given text (positive, negative, or neutral).

Code Example:

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

model = Sequential([
  Embedding(input_dim=10000, output_dim=128),
  LSTM(128),
  Dense(1, activation='sigmoid')
])
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

Emotion Analysis:

Example: Detecting emotions from text (happy, sad, angry, etc.). **Tools:** BERT, RNN, or LSTM networks.

Text Classification:

Example: Categorizing text into predefined categories (e.g., spam vs. non-spam emails).

Aggressive Text Detection:

Example: Using NLP models to classify text as aggressive, hate speech, or neutral.

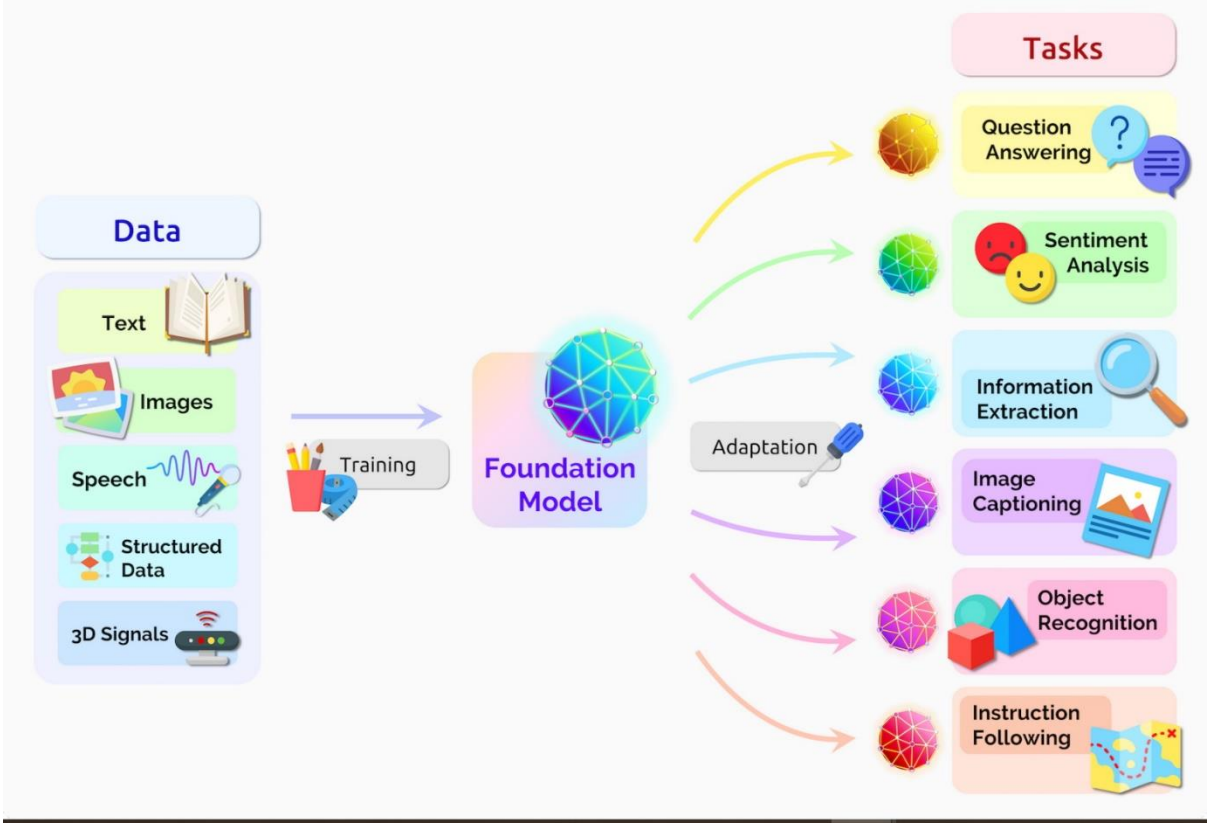
Multimodal Meme Detection:

Example: Classifying memes based on both text and image content (could involve using CNN and RNN models together).

4.2. Interpret Transformer-Based Models

Transformer-Based Model Definition:

A transformer model is a deep learning architecture designed to handle sequence-to-sequence tasks by utilizing self-attention mechanisms. Unlike previous sequence models like RNNs or LSTMs, transformers process the entire input sequence at once.



Why Use Transformer-Based Models?

- **Efficiency:** Transformers can process long-range dependencies more effectively than RNNs.
- **Parallelization:** Unlike RNNs, transformers can be parallelized, leading to faster training times.
- **State-of-the-Art Performance:** Transformers are behind many recent breakthroughs in NLP and computer vision (e.g., BERT, GPT, Vision Transformers).

Transformer vs. Traditional Deep Learning Models:

- **Transformers** use a mechanism called **self-attention** to weigh the importance of each part of the input, making them ideal for sequence tasks.
- **RNNs and LSTMs** process data sequentially, which makes them slower and less effective for longer sequences.

4.3. Design Specific Models

Here are some popular transformer-based models that can be designed and fine-tuned for various NLP tasks:

1. **M-BERT (Multilingual BERT):**
    - Designed to handle multiple languages, M-BERT is pre-trained on a multilingual corpus and fine-tuned for specific tasks.

Example Use Case: Multi-language text classification or sentiment analysis.

2. **Distil-BERT:**

- A smaller, faster, and more efficient version of BERT that retains 97% of BERT's language understanding.

Example Use Case: Text classification with a focus on efficiency and speed.

3. **XML-R (Extreme Multi-label Classification with RoBERTa):**
   - A model optimized for extreme multi-label classification tasks (e.g., assigning multiple categories to a single text).
4. **ROBERTa (Robustly Optimized BERT Pretraining Approach):**
   - A variant of BERT trained with more data, larger batch sizes, and without the Next Sentence Prediction objective.

4.4. Perform Evaluation of DL Models

Performance Metrics:

1. **Accuracy:** Measures how often the model's predictions are correct.
   - Suitable for balanced datasets.
2. **Precision, Recall, F1-Score:**
   - **Precision**: The proportion of true positive predictions to all positive predictions made.
   - **Recall**: The proportion of true positives to all actual positives.
   - **F1-Score**: The harmonic mean of precision and recall. Useful when dealing with imbalanced datasets.

Code Example for Precision, Recall, F1-Score:

```
from sklearn.metrics import classification_report
y_true = [0, 1, 0, 1, 1]  # True labels
y_pred = [0, 1, 0, 0, 1]  # Predicted labels

print(classification_report(y_true, y_pred))
```

3. **Confusion Matrix:**
   - Provides a matrix showing the true positive, true negative, false positive, and false negative predictions.
4. **ROC-AUC Curve:**
   - Visual representation of the model's ability to discriminate between classes.
5. **Loss Function:**
   - Measures the error between predicted and true labels. Common loss functions include Cross-Entropy (for classification) and Mean Squared Error (for regression).

Error Analysis:

Error analysis involves reviewing model predictions to understand where the model is making mistakes. It can be used to identify patterns in misclassifications, leading to better data labeling, feature engineering, and model refinement.

Self-Check Sheet 4: Perform project development activities

Q1: How does a DL model perform handwritten character recognition?

Q2: How is DL used in image classification?

Q3: How does face detection work in DL?

Q4: What is sentiment analysis, and how does DL perform it?

Q5: What is multimodal meme detection?

Q6: Why use transformer-based models?

Q7: How do transformer models differ from traditional DL models?

Q8: What is M-BERT, and where is it used?

Answer Sheet 4: Perform project development activities

Q1: How does a DL model perform handwritten character recognition?
Answer: A DL model, typically a Convolutional Neural Network (CNN), learns patterns in images of handwritten characters. By training on labeled datasets like MNIST, it identifies pixel arrangements that correspond to specific characters, achieving high accuracy in recognizing them in new data.

Q2: How is DL used in image classification?
Answer: DL, especially with CNNs, categorizes images into predefined classes by identifying unique features within each category. For example, a model trained on animals could classify images as “cat,” “dog,” etc., by learning spatial features.

Q3: How does face detection work in DL?
Answer: DL models, often using CNNs or specialized architectures like FaceNet, detect faces by recognizing facial patterns and structures. This technique is widely used in security, photo organization, and augmented reality.

Q4: What is sentiment analysis, and how does DL perform it?
Answer: Sentiment analysis is the process of identifying sentiment (positive, negative, neutral) in text. DL models, like LSTMs or transformers, learn language patterns that convey sentiment, making it useful for customer reviews, social media analysis, and more.

Q5: What is multimodal meme detection?
Answer: Multimodal meme detection identifies harmful content in memes by analyzing both text and images. DL models, often using transformers and CNNs, process visual and textual data together to detect contextual information in multimedia content.

Q6: Why use transformer-based models?
Answer: Transformers excel in handling long-range dependencies and context, providing better performance than traditional RNNs or LSTMs in tasks like NLP. They allow for parallel computation, making them faster and more scalable for large datasets.

Q7: How do transformer models differ from traditional DL models?
Answer: Unlike traditional DL models, which process sequentially (e.g., LSTMs), transformers process sequences in parallel using self-attention, which improves speed and handles context better. This makes transformers more effective for tasks involving long-range dependencies.

Q8: What is M-BERT, and where is it used?
Answer: M-BERT (Multilingual BERT) is a version of BERT trained on multiple languages, making it suitable for cross-lingual NLP tasks like translation, sentiment analysis, and text classification in

Task Sheet 4.1 : Develop a deep learning model for image classification using a convolutional neural network (CNN). Use a dataset of your choice and evaluate the model's performance.

Detailed Steps:

1. **Setup Environment**:
    - Install required libraries:
        - **TensorFlow/Keras** for neural network implementation.
        - **NumPy** for handling data operations.
        - **Pandas** for data handling and manipulation.
        - **Matplotlib/Seaborn** for data visualization (optional).
    - Choose a development environment such as **Jupyter Notebook** or **PyCharm**.
2. **Dataset Selection and Preprocessing**:
    - Choose an appropriate dataset (e.g., **MNIST**, **CIFAR-10**, or a custom dataset).
    - **Preprocess the data**:
        - For images, normalize the pixel values (e.g., scale between 0 and 1).
        - For non-image data, normalize/standardize the features.
        - Split the dataset into **training**, **validation**, and **test** sets.
        - Use **Bootstrap Sampling** (sampling with replacement) to create multiple subsets of the training data. This is crucial for the bagging process.
3. **Define the Base Neural Network Architecture**:
    - Choose a simple neural network architecture (e.g., **Feedforward Neural Network** or **Convolutional Neural Network** if working with images).
        - **For CNN**:
            - Input layer: Process image data (e.g., 28x28 grayscale images for MNIST).
            - Several convolutional layers followed by max-pooling.
            - Flatten the convolutional layer outputs.
            - Fully connected layers.
        - **For Feedforward Neural Network**:
            - Input layer.
            - Several hidden layers with **ReLU activation**.
            - Output layer with **softmax** for classification or **linear activation** for regression tasks.
4. **Bagging Implementation**:
    - **Bootstrap Sampling**: For each model in the ensemble, create a unique bootstrap sample from the training data. Each model will be trained on one of these subsets.
    - **Train multiple instances**:
        - Initialize the same neural network architecture multiple times (e.g., 5 or 10 instances).

        - Train each instance on a different bootstrap sample (randomly drawn with replacement).
    - Ensure that each model is independent by training them in parallel or sequentially, depending on available resources.
5. **Train the Individual Neural Network Models**:
    - Train each instance of the neural network on its respective subset of the training data.
    - Monitor the training process using **early stopping** to avoid overfitting and ensure that each model is properly trained.
    - Use appropriate metrics (e.g., accuracy for classification, MSE for regression) to evaluate performance during training.
6. **Prediction Aggregation**:
    - Once all models in the ensemble have been trained, use them to make predictions on the test data.
    - **For classification tasks**: Use **majority voting** (each model votes for a class, and the most frequent class is chosen as the final prediction).
    - **For regression tasks**: Average the predictions of all models.
7. **Evaluate the Bagging Ensemble**:
    - Compute performance metrics for the ensemble (e.g., accuracy, precision, recall, F1-score for classification or MSE for regression).
    - Compare the performance of the ensemble model with that of a single model trained on the entire dataset.
    - Visualize the results to show how the ensemble model outperforms individual models.
8. **Model Optimization** (if necessary):
    - Tune the hyperparameters (e.g., number of layers, number of neurons in each layer, learning rate) of the base model for better performance.
    - Experiment with the number of models in the ensemble (e.g., try 5 models, 10 models, etc.) to see how it affects performance.
9. **Document Findings and Visualize Results**:
    - Summarize the architecture used for the ensemble and individual models.
    - Present performance comparison in terms of accuracy, loss, and other relevant metrics.
    - Visualize the model's performance using plots such as training curves, confusion matrix, or bar charts for comparison.

Reference

1. **Deep Learning with Python** - François Chollet (2017)
2. **Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow** - Aurélien Géron (2019)
3. **Neural Networks and Deep Learning: A Textbook** - Charu Aggarwal (2018)
4. **Deep Learning for Computer Vision** - Rajalingappaa Shanmugamani (2018)
5. **AI for Games, Second Edition** - Ian Millington (2019)
6. **Practical Deep Learning for Coders** - Jeremy Howard, Sylvain Gugger (2020)
7. **Deep Learning: A Practitioner's Approach** - Adam Gibson, Josh Patterson (2017)
8. **Transformers for Natural Language Processing** - Denis Rothman (2021)
9. **Augmented Reality: Principles and Practice** - Dieter Schmalstieg, Tobias Hollerer (2016)
10. **Deep Learning with TensorFlow 2 and Keras** - Antonio Gulli, Amita Kapoor, Sujit Pal (2020)

NB: After completion of all LO, then complete the following review of competency

Review of Competency

Below is yourself assessment rating for module "Implement of Deep Learning"

Assessment of performance Criteria	Yes	No
1. Fundamentals of DL is interpreted	☐	☐
2. Activities for DL Tools and libraries are performed	☐	☐
3. Data preparation is implemented	☐	☐
4. Manual and automatic data labeling are interpreted	☐	☐
5. Automatic labeling techniques are implemented	☐	☐
6. Feature extraction is implemented	☐	☐
7. Visualization of word vectors are implemented with word cloud, histogram, heatmap, plots and tableau	☐	☐
8. Embedding models are exercised	☐	☐
9. Pre-trained word embedding is implemented	☐	☐
10. ANN and CNN are comprehended	☐	☐
11. CNN Variations are implemented	☐	☐
12. Optimization of hyperparameters is implemented	☐	☐
13. Recurrent neural networks are implemented	☐	☐
14. Ensemble of DL Models is implemented	☐	☐
15. Projects are developed using DL	☐	☐
16. Transformer based models are interpreted	☐	☐
17. Specific models are designed	☐	☐
18. Evaluation of DL models are performed	☐	☐

I now feel ready to undertake my formal competency assessment.

Signed:

Date:

Development of CBLM

The Competency based Learning Material (CBLM) of 'Implement of Deep Learning' (Occupation: AI in Immersive Technology) for National Skills Certificate is developed by NSDA with the assistance of SAMAHAR Consultants Ltd.in the month of June, 2024 under the contract number of package SD-9C dated 15th January 2024.

SL No.	Name and Address	Designation	Contact Number
1	A K M Mashuqur Rahman Mazumder	Writer	Cell: 01676323576 Email : mashuq.odelltech@odell.com.bd
2	Nafija Arbe	Editor	Cell: 01310568900 nafija.odelltech@odell.com.bd
3	Khan Mohammad Mahmud Hasan	Co-Ordinator	Cell: 01714087897 Email: kmmhasan@gmail.com
4	Md. Saif Uddin	Reviewer	Cell: 01723004419 Email:engrbd.saif@gmail.com