TITLE: Parallel Sorting Algorithms

📚 **THEORY EXPLAINED (What is OpenMP?):**

- OpenMP is an **API and set of compiler directives** used in C, C++, and Fortran to perform **parallel programming** on shared-memory machines.
- When we write #pragma omp parallel, the compiler creates **multiple threads**, typically equal to the number of processor cores.
- These threads **execute the same block of code simultaneously**, improving speed when applied properly.
- Commonly used directives:
    - #pragma omp parallel: Creates threads.
    - #pragma omp for: Splits loop iterations among threads.
    - #pragma omp barrier: Synchronizes threads.
    - #pragma omp sections: Runs different code blocks in parallel.

🔍 **PROGRAM 1: Bubble Sort (Sequential and Parallel)**

✅ **Function 1: sBubble()**

- This is the **Sequential Bubble Sort**.
- It uses a nested loop to compare adjacent elements and **swap** if they are in the wrong order.
- Time complexity: **$O(n^2)$**.

✅ **Function 2: pBubble()**

- This is the **Parallel Bubble Sort** using OpenMP.
- The sorting is done in **two phases per iteration**:
    - **Odd Phase**: Compares and swaps arr[1] & arr[2], arr[3] & arr[4], etc.
    - **Even Phase**: Compares and swaps arr[0] & arr[1], arr[2] & arr[3], etc.
- OpenMP parallelizes each phase using #pragma omp for.
- A **barrier** ensures all threads complete one phase before moving to the next.

✅ **Function 3: printArray()**

- Used to display the array after sorting.

✅ **Main function:**

- Initializes an array in reverse order (worst case).
- First runs and times the **sequential sort**, then the **parallel sort**, and prints both times.

🔍 **PROGRAM 2: Merge Sort (Sequential and Parallel)**

✅ **Function 1: merge()**

- Merges two **sorted subarrays** into one.
- This is a helper used by both sequential and parallel merge sort.
- Ensures **stable and efficient merging** in O(n) time.

✅ **Function 2: mergeSort()**

- This is the **Sequential Merge Sort**.
- Recursively divides the array and calls merge() to combine.
- Time complexity: **O(n log n)**.

✅ **Function 3: parallelMergeSort()**

- This is the **Parallel Merge Sort using OpenMP**.
- Uses #pragma omp parallel sections to **sort two halves of the array in parallel**.
- After sorting both halves, it merges them.
- Exploits **divide-and-conquer** for parallelism.

✅ **Main function:**

- Initializes the array.
- Runs and times the **sequential merge sort**.
- Reinitializes the array and times the **parallel merge sort**.

🧠 ORAL VIVA QUESTIONS AND ANSWERS

---

1. Do the analysis of parallel bubble sort and find out its time complexity.

- **Parallel bubble sort** still has **$O(n^2)$** time complexity.
- But parallelizing **odd and even phases** improves performance in practice by **reducing wall-clock time**.
- It doesn't reduce the algorithmic complexity but **improves efficiency** with multiple threads.

---

2. Write the difference between parallel approach bubble sort and merge sort.

Feature	**Parallel Bubble Sort**	**Parallel Merge Sort**
Basic Method	Swap adjacent elements	Divide array and merge sorted subarrays
Time Complexity	$O(n^2)$	O(n log n)
Parallelism Approach	Parallel odd/even passes	Divide-and-conquer using recursive calls
Efficiency with threads	Limited scaling	Scales well with number of threads
Use of OpenMP	#pragma omp for, barrier	#pragma omp sections

---

3. Comment on scaling parallel merge sort.

- Parallel merge sort **scales better** than bubble sort.
- As the array size grows, **more subarrays** can be sorted in parallel.
- The time to sort reduces significantly with **multiple cores**.
- The overhead is less because merge sort **divides work evenly** and performs **independent tasks**.

---

4. How can the bubble sort algorithm be parallelized?

- By alternating between **odd and even phases**, we can sort pairs simultaneously.
- Use #pragma omp for to divide work among threads.

- Add a #pragma omp barrier between phases to **synchronize** threads.
- This works well for small arrays, but due to its $O(n^2)$ nature, not ideal for large-scale parallelism.

Title: Parallel DFS and BFS using OpenMP

✅ **Key Concepts**

🔸 **DFS (Depth-First Search):**

- DFS goes deep into the graph/tree by visiting child nodes before siblings.
- Uses **stack (implicit via recursion or explicit)**.
- Ideal for **cycle detection**, **topological sorting**, **connected components**.

🔸 **BFS (Breadth-First Search):**

- BFS explores all neighbors first, then their children.
- Uses a **queue**.
- Good for **finding shortest path in unweighted graphs**, **level-order traversal**, etc.

🔸 **OpenMP:**

- A parallel programming API for C/C++/Fortran.
- We use #pragma omp parallel for to parallelize loops.

✅ **Code Explanation**

🔹 **Parallel BFS Code (using Queue)**

```
#include<iostream>

#include<queue>

#include<vector>
```

```
#include<omp.h>
using namespace std;
```

📌 **Purpose:**

- Includes headers for standard I/O, graph data structure, and OpenMP.

```
int num_vertices, num_edges, source;
cout<<"Enter total No. of vertices, Edges and start vertex: "<<endl;
cin>>num_vertices>>num_edges>>source;
```

📌 **Purpose:**

- Take inputs from user: number of vertices, edges, and the starting node.

```
vector<vector<int>> adj_list(num_vertices+1);
```

📌 **Purpose:**

- Create adjacency list for the graph.

```
for (int i = 0; i < num_edges; i++) {
   int u, v;
   cin >> u >> v;
   adj_list[u].push_back(v);
   adj_list[v].push_back(u);
}
```

📌 **Purpose:**

- Read the undirected graph edges and populate the adjacency list.

```
queue<int> q;
vector<bool> visited(num_vertices + 1, false);
```

```
q.push(source);
visited[source] = true;
```

📌 **Purpose:**

- Initialize BFS with queue and mark source node as visited.

```
while (!q.empty()) {
    int curr_vertex = q.front();
    q.pop();
    cout << curr_vertex << " ";

    #pragma omp parallel for shared(adj_list, visited, q) schedule(dynamic)
    for (int i = 0; i < adj_list[curr_vertex].size(); i++) {
        int neighbour = adj_list[curr_vertex][i];
        if (!visited[neighbour]) {
            visited[neighbour] = true;
            q.push(neighbour);
        }
    }
}
```

📌 **Purpose:**

- BFS traversal using a queue.
- #pragma omp parallel for: parallelizes the for-loop of neighbors.
- **Limitation**: Directly pushing to queue from multiple threads is unsafe in real OpenMP; this is for academic understanding only.

🔹 **Parallel DFS Code (using Recursion)**

```
vector<int> adj[MAXN + 5];
bool visited[MAXN + 5];
```

📌 **Purpose:**

- Adjacency list and visited array (global) for DFS.

```
void dfs(int node) {
  visited[node] = true;

  #pragma omp parallel for
  for (int i = 0; i < adj[node].size(); i++) {
    int next_node = adj[node][i];
    if (!visited[next_node]) {
      dfs(next_node);
    }
  }
}
```

📌 **Purpose:**

- DFS function using recursion.
- Parallel for loop tries to explore multiple neighbors in parallel.
- **Note**: Recursion + OpenMP isn't always ideal due to stack safety; still, for learning it's fine.

```
int main(){
  int n, m, sNode;
  cin>>n>>m;
  for (int i = 1; i <= m; i++) {
```

```
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    cin >> sNode;
    dfs(sNode);

    for (int i = 1; i <= n; i++) {
        if (visited[i]) {
            cout << i << " ";
        }
    }
    return 0;
}
```

📌 **Purpose:**

- Builds the graph, calls DFS, then prints visited nodes.

✅ **Viva/Oral Questions & Answers**

Q1: What is the difference between DFS and BFS?

- **DFS**: Goes deep along each path. Uses **stack or recursion**.
- **BFS**: Visits level by level. Uses a **queue**.

Q2: How is OpenMP used in your code?

- #pragma omp parallel for is used to explore adjacent nodes of the current node in parallel.

Q3: What are the limitations of using parallelism in BFS/DFS?

- Race conditions if shared data structures like queue or visited are not protected.
- DFS is recursive—parallel recursion can lead to stack issues.
- BFS queue is not thread-safe—requires mutex or other synchronization for true parallel execution.

Q4: Why use parallelism here?

- Graphs with large branching factor can benefit from exploring multiple neighbors simultaneously.
- Reduces traversal time especially for wide/deep graphs.

Title: Parallel Reduction

📘 **Theory:**

🔹 **What is Parallel Reduction?**

Parallel Reduction is the process of combining elements (like summing all values in a list) in **parallel**, instead of sequentially. This is important in:

- Scientific computing
- Machine learning
- Big data analysis

In reduction, a large array or vector is processed to **reduce** it to a **single result** (e.g., sum, min, max).

🔹 **What is CUDA?**

CUDA (Compute Unified Device Architecture) is a parallel programming platform by **NVIDIA** used to accelerate computations using **GPUs**. Although CUDA is for GPU programming, **in this assignment, OpenMP is used**, which is for **multi-core CPU parallelism**.

⚙️ **Working of the Code (Using OpenMP):**

The program uses **4 separate functions** to perform the reductions:

vector<int> arr = {5, 2, 9, 1, 7, 6, 8, 3, 4};

🧠 **FUNCTION-WISE ORAL EXPLANATION:**

🔹 **1. min_reduction(vector<int>& arr)**

```
int min_value = INT_MAX;
#pragma omp parallel for reduction(min: min_value)
for (int i = 0; i < arr.size(); i++) {
   if (arr[i] < min_value) {
      min_value = arr[i];
   }
}
```

- **Purpose:** Find the **minimum value** in the vector.
- **reduction(min: min_value)**: OpenMP keeps local min_value copies in threads and returns the **minimum** after all threads finish.
- **INT_MAX**: Initialized to a very large number so any value in the array will be smaller.
- **Output:** "Minimum value: 1"

🗣️ *In oral*: “This function calculates the minimum element from a large array using OpenMP's reduction(min:) clause which reduces local thread minimums to a global minimum.”

🔹 **2. max_reduction(vector<int>& arr)**

```
int max_value = INT_MIN;
#pragma omp parallel for reduction(max: max_value)
for (int i = 0; i < arr.size(); i++) {
```

```
        if (arr[i] > max_value) {
            max_value = arr[i];
        }
    }
```

- **Purpose:** Find the **maximum value** in the vector.
- **reduction(max:)** ensures the largest value is selected across all threads.
- **INT_MIN** initializes to lowest possible integer to ensure comparison works.
- **Output:** "Maximum value: 9"

🗣 *In oral*: “This function finds the maximum element using the reduction(max:) directive of OpenMP.”

🔹 **3. sum_reduction(vector<int>& arr)**

```
    int sum = 0;
    #pragma omp parallel for reduction(+: sum)
    for (int i = 0; i < arr.size(); i++) {
        sum += arr[i];
    }
```

- **Purpose:** Compute the **sum** of the elements.
- **reduction(+: sum)** means each thread adds up its portion, and OpenMP combines them.
- **Output:** "Sum: 45"

🗣 *In oral*: “This function calculates the sum of all elements using OpenMP's reduction(+: sum) directive for efficient parallel computation.”

🔹 **4. average_reduction(vector<int>& arr)**

```
    int sum = 0;
    #pragma omp parallel for reduction(+: sum)
```

```
for (int i = 0; i < arr.size(); i++) {
    sum += arr[i];
}
cout << "Average: " << (double)sum / arr.size() << endl;
```

- **Purpose:** Compute **average** by first finding the sum (same as previous function).
- **Casts to double** to ensure decimal precision.
- **Output:** "Average: 5.0"

🗣 *In oral*: “This function reuses the sum logic to find average by dividing the total sum by the size of the array.”

🏁 MAIN FUNCTION main()

```
vector<int> arr = {5, 2, 9, 1, 7, 6, 8, 3, 4};
```

- A sample vector is declared and passed to all four functions:
  - min_reduction(arr)
  - max_reduction(arr)
  - sum_reduction(arr)
  - average_reduction(arr)

🗣 *In oral*: “The main function initializes a static vector and calls all four reduction functions to demonstrate OpenMP parallelism.”

❓ QUESTIONS FOR ORAL EXAM (Suggested Answers):

Q1. How is CUDA programming useful to study parallel algorithms?

Ans: CUDA helps us utilize GPU’s parallel cores to solve problems faster by executing multiple tasks simultaneously. It is especially useful for data-intensive tasks like vector/matrix operations, image processing, etc.

Q2. Importance of parallel reduction operations?

Ans: Parallel reduction allows us to compute results like min, max, or sum faster by distributing the workload across multiple cores instead of sequentially looping through large datasets.

Q3. Discuss operations on vectors and parallel algorithm design.

Ans: Operations like finding min, max, sum, or average can be divided among threads. Using OpenMP or CUDA, each thread processes a portion of data and then merges results (reduction).

Q4. How is parallelism achieved in CUDA?

Ans: In CUDA, parallelism is achieved using thousands of **threads**, which are organized into **blocks** and **grids**. Each thread executes the same code on different data in parallel.

Q5. Explain Grid, Block, and Thread in parallel reduction.

Ans:

- **Grid:** Collection of blocks.
- **Block:** Collection of threads.
- **Thread:** Smallest unit of execution. Each thread performs operations on one or more data elements.
  In reduction, each thread processes part of the array and then results are combined.

TITLE: Vector and Matrix Operations using CUDA

🔸 **PART 1: Vector Addition using CUDA**

✅ **Concept:**

Vector addition is an element-wise operation:
If A = [a1, a2, a3], B = [b1, b2, b3], then C = [a1 + b1, a2 + b2, a3 + b3].

✅ **Steps in CUDA Program:**

1. Header and Kernel Definition:

cpp

CopyEdit

__global__ void add(int* A, int* B, int* C, int size)

- This is the **GPU kernel function**.
- Each **thread** adds one element: C[tid] = A[tid] + B[tid].

2. Initialize and Allocate Host Memory:

cpp

CopyEdit

A = new int[vectorSize]; // Host memory for input vectors

3. Initialize Values:

cpp

CopyEdit

void initialize(int* vector, int size)

- Fills vector A and B with random values using rand().

4. Device Memory Allocation:

cpp

CopyEdit

cudaMalloc(&X, vectorBytes);

- Allocates GPU memory for vectors A, B, C.

5. Copy from Host to Device:

cpp

CopyEdit

cudaMemcpy(X, A, vectorBytes, cudaMemcpyHostToDevice);

6. Kernel Launch:

cpp

CopyEdit

add<<<blocksPerGrid, threadsPerBlock>>>(X, Y, Z, N);

- <<<grid, block>>> is CUDA syntax to define how many threads to launch.

7. Copy Result Back to Host:

cpp

CopyEdit

cudaMemcpy(C, Z, vectorBytes, cudaMemcpyDeviceToHost);

8. Free Memory (Clean Up):

cpp

CopyEdit

delete[] A; cudaFree(X);

✅ **Sample Output:**

yaml

CopyEdit

Vector A: 3 6 7 5

Vector B: 3 5 6 2

Addition: 6 11 13 7

🔸 **PART 2: Matrix Multiplication using CUDA**

✅ **Concept:**

If A is NxN and B is NxN,
then C[i][j] = sum of (A[i][k] * B[k][j]) for k from 0 to N.

✅ **Steps in CUDA Program:**

1. Matrix Initialization:

cpp

CopyEdit

```
for (int j = 0; j < N; j++)
   for (int i = 0; i < N; i++)
      hA[j*N + i] = 2; hB[j*N + i] = 4;
```

- All values of A and B are filled with constants 2 and 4 for simplicity.

2. Device Memory Allocation:

cpp

CopyEdit

```
cudaMalloc(&dA, size); // GPU memory
```

3. Kernel Definition:

cpp

CopyEdit

```
__global__ void gpuMM(float *A, float *B, float *C, int N)
```

- Each thread calculates **one element** of the result matrix C.

cpp

CopyEdit

```
int row = blockIdx.y*blockDim.y + threadIdx.y;
int col = blockIdx.x*blockDim.x + threadIdx.x;
```

- Loop for (int n = 0; n < N; ++n) performs the dot product.

4. Thread Block and Grid Setup:

cpp

CopyEdit

```
dim3 threadBlock(BLOCK_SIZE,BLOCK_SIZE);
dim3 grid(K,K);
```

5. Kernel Launch:

cpp

CopyEdit

gpuMM<<<grid, threadBlock>>>(dA, dB, dC, N);

6. Copy Result Back to Host:

cpp

CopyEdit

cudaMemcpy(C, dC, size, cudaMemcpyDeviceToHost);

7. Display Output:

cpp

CopyEdit

cout << C[row * col] << " ";

(Note: should ideally be C[row * N + col] to avoid bugs.)

✅ **Sample Output:**

yaml

CopyEdit

Input Matrix 1:

2 2

2 2

Input Matrix 2:

4 4

4 4

Resultant matrix:

16 16

16 16

Explanation:
Each element is computed as:
(2*4 + 2*4) = 8 + 8 = 16

🔸 **VIVA QUESTIONS WITH ANSWERS**

❓ **Q1. What are the advantages of using CUDA over CPU?**

✅ **Answer:**

- CUDA allows **parallel execution** of thousands of threads.
- It **reduces computation time** drastically for large data.
- Ideal for **vector/matrix operations, ML, graphics**.
- GPUs have more **arithmetic units**, making them more suitable for these tasks.

❓ **Q2. How do you launch a CUDA kernel?**

✅ **Answer:**
Using the syntax:

cpp

CopyEdit

kernelName<<<number_of_blocks, threads_per_block>>>(args);

For example:

cpp

CopyEdit

add<<<blocksPerGrid, threadsPerBlock>>>(X, Y, Z, N);

This tells the GPU how many parallel threads to launch.

❓ **Q3. How can you optimize performance?**

✅ **Answer:**

1. Use **Shared Memory** for fast data access.
2. Adjust **block and grid sizes** to maximize GPU occupancy.
3. Avoid **memory bottlenecks** by minimizing host-device transfers.
4. Use **coalesced memory access** (aligned reads/writes).
5. Minimize **branching (if-else)** inside kernels.

..............DL..............

1] Linear regression

Dataset Information:

The **Boston Housing Dataset** contains **506 samples** and **13 features** (or independent variables), with the target variable being **MEDV** – the Median value of owner-occupied homes in $1000s.

Some of the important features are:

- RM – Average number of rooms per dwelling.
- LSTAT – % of lower status of the population.
- CRIM – Crime rate per capita.
- NOX – Nitric oxide concentration.
- ...and so on.

The target is a **continuous value**, so we apply **regression**, not classification.

📘 Theory Concepts Used:

1. **Linear Regression**
    - Predicts a continuous numeric value using a linear relationship between inputs and the output.
    - Simple form: y = mx + c
2. **Deep Neural Network (DNN)**
    - A neural network with multiple **hidden layers**.
    - Each layer has **neurons** with **activation functions**.
    - In this assignment, we use ReLU for hidden layers and Linear for the output.
3. **Standardization (Scaling)**
    - Important to normalize features to bring them on the same scale.
    - We use **MinMaxScaler** to scale data between 0 and 1.
4. **Train-Test Split**
    - Data is split into **training set** (70%) and **testing set** (30%).
    - Training set is used to fit the model.
    - Testing set is used to evaluate model performance.

🧠 Step-by-step Code and Function Explanation:

🔹 1. Loading the Dataset

```
df=pd.read_csv("1_boston_housing.csv")
```

- Loads the dataset from CSV using pandas.

```
print(df.head())        # Shows first 5 rows
print(df.shape)         # Returns (506, 14)
print(df.size)          # Total number of elements (rows * columns)
print(df.ndim)          # Dimension of dataframe (2)
print(df.dtypes)        # Data types of each column
df.describe()           # Statistical summary of numerical columns
df.columns              # Column names
df.info()               # Overall info like non-null counts, data types
```

🔹 2. Preparing Input and Output Variables

```
x = df.loc[:, df.columns != 'MEDV']   # All features (13 columns)
y = df.iloc[:, df.columns == 'MEDV']  # Target/label column (house price)
```

🔹 3. Splitting Dataset

```
from sklearn.model_selection import train_test_split
```

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=123)

- Splits data into 70% training and 30% testing.
- random_state=123 ensures reproducibility.

🔹 4. Scaling the Features

```
from sklearn.preprocessing import MinMaxScaler

mms = MinMaxScaler()

mms.fit(x_train)

x_train = mms.transform(x_train)

x_test = mms.transform(x_test)
```

- Scales input features to 0–1 range using **MinMaxScaler**.
- **fit** learns min/max from training data.
- **transform** applies scaling to both train and test sets.

🔹 5. Building the Deep Neural Network

```
from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense


model = Sequential()
```

model.add(Dense(128, input_shape=(13,), activation='relu', name='dense_1'))

model.add(Dense(64, activation='relu', name='dense_2'))

model.add(Dense(1, activation='linear', name='dense_output'))

- We use **Keras** with TensorFlow backend.
- Sequential means stacking layers one after another.
- **Layer 1:** 128 neurons, ReLU activation.
- **Layer 2:** 64 neurons, ReLU.
- **Output Layer:** 1 neuron, Linear activation (for regression).

🔹 6. Compiling the Model

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

- **Optimizer:** adam is used for fast gradient descent.
- **Loss Function:** Mean Squared Error (MSE) is suitable for regression.
- **Metric:** MAE (Mean Absolute Error).

🔹 7. Training the Model

history = model.fit(x_train, y_train, epochs=100, validation_split=0.05, verbose=1)

- Trains the model for **100 epochs**.
- Uses 5% of training data for **validation**.

- verbose=1 shows training progress.

🔹 8. Evaluating the Model

```
mse_nn, mae_nn = model.evaluate(x_test, y_test)

print('Mean squared error on test data: ', mse_nn)

print('Mean absolute error on test data: ', mae_nn)
```

- Evaluates on test set and prints **MSE** and **MAE**.

🔹 9. Making Predictions and Final Metrics

```
from sklearn.metrics import mean_squared_error,
mean_absolute_error

y_pred = model.predict(x_test)


mse = mean_squared_error(y_test, y_pred)

mae = mean_absolute_error(y_test, y_pred)


print(f"Mean Squared Error (MSE): {mse:.2f}")

print(f"Mean Absolute Error (MAE): {mae:.2f}")
```

- Uses sklearn to double-check evaluation metrics.
- y_pred is predicted values; compared with y_test.

✅ Conclusion:

We successfully implemented **Linear Regression using a Deep Neural Network** on the **Boston Housing Dataset**.
The model can now predict house prices based on input features like room count, crime rate, etc.

🧠 Oral Answers to Viva Questions:

1. **What is Linear Regression?**
   It is a statistical method that models the relationship between a dependent and one or more independent variables using a straight line.
2. **What is a Deep Neural Network?**
   A machine learning model with multiple hidden layers of neurons that can learn complex patterns in data.
3. **What is the concept of standardization?**
   Standardization scales data to a fixed range or distribution. In our case, MinMaxScaler converts features to a 0–1 range for better model performance.
4. **Why split data into train and test?**
   To evaluate the model's performance on unseen data and check for overfitting or underfitting.
5. **Applications of DNN:**
   - Image and speech recognition
   - Natural Language Processing

- Medical diagnosis
- Fraud detection
- Price prediction

ASSIGNMENT AIM:

To perform:

1. **Multiclass classification** using a **Deep Neural Network (DNN)** on the **OCR letter recognition dataset**.
2. **Binary classification** using a **DNN** on **IMDB movie reviews** for sentiment analysis (positive/negative).

✅ **PART 1: Binary Classification using IMDB dataset**

We classify movie reviews as **positive** or **negative** using a **Deep Neural Network** built with **TensorFlow/Keras**.

➤ **Code Explanation Line-by-Line:**

python

CopyEdit

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers

```
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences
import matplotlib.pyplot as plt
```

- We import **TensorFlow and Keras** libraries for deep learning.
- imdb provides the dataset.
- pad_sequences helps standardize input length.
- matplotlib.pyplot is for plotting accuracy/loss graphs.

```
python
CopyEdit
vocab_size=10000
maxlen=200
```

- Limit vocabulary to **10,000 most frequent words**.
- Set **maximum review length** to 200 words for padding.

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

- Load pre-tokenized IMDB dataset. Reviews are already converted into sequences of integers.
- x_train, x_test are input reviews; y_train, y_test are corresponding sentiment labels (0 or 1).

```
python
```

CopyEdit

x_train=pad_sequences(x_train,maxlen=maxlen)

x_test=pad_sequences(x_test,maxlen=maxlen)

- Pads all sequences to same length (200). Short reviews are padded with zeros.

➤ **Model Building**

python

CopyEdit

model=keras.Sequential([

layers.Embedding(input_dim=vocab_size,output_dim=32,input_length=maxlen),

layers.GlobalAveragePooling1D(),

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

layers.Dense(1,activation='sigmoid')

])

- **Embedding layer**: Converts integer tokens to dense vectors of size 32.
- **GlobalAveragePooling1D**: Averages the embedding vectors to a fixed-size vector.
- **Dense(64, relu)**: Fully connected layer with 64 neurons.

- **Dense(1, sigmoid)**: Output layer for binary classification. Returns a probability between 0 and 1.

➤ Model Compilation and Training

python

CopyEdit

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

- **Adam**: Adaptive optimizer.
- **Binary Crossentropy**: Used for binary classification.
- **Accuracy**: Performance metric.

python

CopyEdit

```
history=model.fit(x_train,y_train,epochs=100,batch_size=512,validation_split=0.2,verbose=1)
```

- Train for **100 epochs**.
- **512 samples per batch**.
- Use 20% of training data as validation.
- verbose=1 shows training progress.

➤ Model Evaluation and Visualization

python

CopyEdit

```
loss,accuracy=model.evaluate(x_test,y_test,verbose=1)
print(f"test accuracy:{accuracy:.4f}")
```

- Evaluate model on **test data**.
- Print final **accuracy**.

➤ Plotting Accuracy and Loss

python

CopyEdit

```
plt.subplot(1, 2, 1)
# Accuracy plot
plt.subplot(1, 2, 2)
# Loss plot
```

- Shows training vs validation performance over epochs.

➤ Prediction & Interpretation

python

CopyEdit

```
y_pred_probs=model.predict(x_test[:10])
y_pred_classes=(y_pred_probs > 0.5).astype("int32")
```

- Predict on 10 test reviews.

- Convert probabilities to class labels: if prob > 0.5 → positive.

python

CopyEdit

```
for i in range(10):
    print(f"Review {i+1} - predict: {'positive' if y_pred_classes[i][0]==1 else 'Negative'},Actual: {'Positive' if y_test[i]==1 else 'Negative'}")
```

- Print predicted and actual labels for first 10 reviews.

✅ **PART 2: Multiclass Classification using OCR Dataset**

You classify letters A–Z (26 classes) using **MLPClassifier** (Multi-layer Perceptron) from **scikit-learn**.

➤ **Code Explanation:**

python

CopyEdit

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import model_selection, preprocessing
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report, confusion_matrix as cm1, accuracy_score
```

- **Pandas**: for reading and manipulating the dataset.

- **MLPClassifier**: For deep learning model (multiclass classification).
- **StandardScaler**: Normalize features.
- **Metrics**: To evaluate model accuracy and classification results.

➤ **Dataset Loading**

python

CopyEdit

dataset=pd.read_csv("letter-recognition.data",sep=",")

- Load the OCR dataset which contains 20,000 samples and 17 columns (1 label + 16 features).

python

CopyEdit

x=dataset.iloc[:,1:17] # Input features

y=dataset.select_dtypes(include=[object]) # Output labels (letters)

- x contains numerical features.
- y contains the target labels (A–Z).

➤ **Data Preprocessing**

python

CopyEdit

scaler=StandardScaler()

scaler.fit(x_train)

x_train=scaler.transform(x_train)

x_validation=scaler.transform(x_validation)

- Normalize input data to zero mean and unit variance.
- Important for training neural networks efficiently.

➤ Model Training

python

CopyEdit

mlp=MLPClassifier(hidden_layer_sizes=(250,300),max_iter=1000000,activation='logistic')

- **MLPClassifier** builds a deep neural net with:
    - Hidden layers: 2 layers with 250 and 300 neurons.
    - Activation: 'logistic' (sigmoid).
    - Max iterations: 1 million for convergence.

➤ Confusion Matrix (Yellowbrick Visualization)

python

CopyEdit

from yellowbrick.classifier import confusion_matrix

cm=confusion_matrix(mlp,x_train,y_train,x_validation,y_validation,classes="A,B,...,Z".split(','))

- Visualizes model predictions against actual labels (confusion matrix) for 26 classes A–Z.

python

CopyEdit

cm.fit(x_train,y_train.values.ravel())

cm.score(x_validation,y_validation)

- Train the model and calculate validation accuracy.

➤ **Prediction & Accuracy**

python

CopyEdit

predictions=cm.predict(x_validation)

print("Accuracy= ",accuracy_score(y_validation,predictions))

- Predict letter classes.
- Print overall classification accuracy.

📊 **Performance Metrics Used**

1. **Accuracy**: Overall percentage of correct predictions.
2. **Precision**: Correct positive predictions over total predicted positives.

3. **Recall**: Correct positive predictions over total actual positives.
4. **F1-Score**: Harmonic mean of precision and recall.

✅ THEORY/VIVA QUESTIONS

Q1: What is Multiclass Classification?
Multiclass classification is a machine learning task where the goal is to assign an input to one of **three or more** classes.
Example: Classifying handwritten letters A to Z (26 classes).

Q2: What is Binary Classification with an Example?
Binary classification involves classifying input data into one of **two categories**.
Example: IMDB movie review classification → Predict whether a review is "Positive" (1) or "Negative" (0).

Assignment No: 3 – Convolutional Neural Networks (CNN)

📚 THEORY (Oral Explanation)

🌐 What is a CNN (Convolutional Neural Network)?

A CNN is a **deep learning algorithm** designed to automatically and adaptively learn **spatial hierarchies of features** from images. It uses:

- **Convolution layers** (to extract features)
- **Pooling layers** (to reduce dimensionality)

- **Fully connected layers** (for final classification)

🧪 EXPERIMENT: CNN for Fashion MNIST

✅ 1. Importing Required Libraries

python

CopyEdit

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import numpy as np
```

- tensorflow/keras: For building and training the CNN model.
- matplotlib.pyplot: To plot training graphs and image predictions.
- numpy: To handle numeric data.

✅ 2. Loading the Fashion MNIST Dataset

python

CopyEdit

```
fashion_mnist = keras.datasets.fashion_mnist
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
```

- fashion_mnist: Built-in dataset containing **70,000 grayscale images** (28x28 pixels) of **10 clothing categories**.
- x_train, y_train: Training images and labels (60,000 samples).
- x_test, y_test: Testing images and labels (10,000 samples).

✅ 3. Normalizing the Data

python

CopyEdit

```
x_train = x_train / 255.0

x_test = x_test / 255.0
```

- Pixel values range from 0 to 255.
- Dividing by 255 scales values to [0, 1], which **improves convergence** during training.

✅ 4. Defining Class Names

python

CopyEdit

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',

               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

These are **human-readable labels** for each class (used during prediction visualization).

✅ 5. Building the CNN Model

python

CopyEdit

```
model = keras.Sequential([
    layers.Reshape((28, 28, 1), input_shape=(28, 28)),
    layers.Conv2D(32, (3, 3), activation='relu'),
    layers.MaxPooling2D(2, 2),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D(2, 2),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

- **Explanation of Layers:**
  1. Reshape: Converts input shape from (28,28) to (28,28,1) for grayscale images.
  2. Conv2D(32, (3,3), relu): Applies 32 filters of size 3x3 to learn basic features (edges, textures).
  3. MaxPooling2D(2,2): Downsamples the feature map to reduce computation.
  4. Conv2D(64, (3,3), relu): Learns more complex patterns.
  5. MaxPooling2D(2,2): Further downsampling.

6. Flatten(): Flattens the 2D output to a 1D vector.
7. Dense(128, relu): Fully connected layer to learn non-linear combinations.
8. Dense(10, softmax): Output layer for 10 categories. Softmax converts outputs to probabilities.

✅ 6. Compiling the Model

python

CopyEdit

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

- **Adam**: Adaptive optimizer for better and faster convergence.
- **Loss**: For multiclass classification, sparse_categorical_crossentropy is used.
- **Metric**: Accuracy is used to measure performance.

✅ 7. Training the Model

python

CopyEdit

```
history = model.fit(x_train, y_train, epochs=10, validation_split=0.2)
```

- Trains model for 10 epochs (10 full passes over the training set).
- Uses 20% of training data for **validation**.

- history stores loss and accuracy for each epoch.

✅ 8. Evaluating the Model

python

CopyEdit

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc:.4f}")
```

- Model is tested on **unseen data** (x_test).
- Accuracy and loss are printed.

✅ 9. Plotting Training History

python

CopyEdit

```
plt.plot(...)  # Accuracy plot
plt.plot(...)  # Loss plot
```

- These plots help us **visualize overfitting/underfitting**.
- Useful to analyze model performance epoch-wise.

✅ 10. Making Predictions

python

CopyEdit

predictions = model.predict(x_test[:5])

- Predicts classes for first 5 images in the test set.

python

CopyEdit

for i in range(5):

plt.imshow(...)

plt.title(...)

- Displays the test image.
- Shows predicted and actual label.

✅ **CONCLUSION (Oral Answer):**

We successfully implemented a CNN model that classifies images from the Fashion MNIST dataset. The CNN uses convolution and pooling layers to automatically extract features and uses fully connected layers to classify the images into 10 categories.

❓ **VIVA QUESTIONS – Suggested Answers**

1. **What is a CNN?**
    - CNN stands for Convolutional Neural Network. It's a type of deep learning model designed to work with images. It uses convolutional layers to extract features and pooling layers to reduce dimensionality.

2. **Explain LeNet5 and AlexNet (any one from the list):**
    - LeNet5 is one of the earliest CNN architectures (by Yann LeCun). It has two convolution layers and three fully connected layers.
    - AlexNet is a deeper CNN that won the ImageNet competition. It uses ReLU activation, dropout, and data augmentation to improve performance.

Recurrent neural network (RNN)

Introduction

In this practical, we used a **Recurrent Neural Network (RNN)** model to perform **time series prediction** of Google's stock prices.
Time series prediction means we predict future values based on previous trends.
Since stock prices are sequential and depend on past values, RNN is well-suited because it can **remember patterns from the past using its memory cells**.

🧠 Theory: What is an RNN?

- **RNN** stands for Recurrent Neural Network. It is a type of neural network designed for **sequential or time-series data**.

- It processes input one step at a time and **remembers previous steps using internal memory** (called hidden states).
- It is different from normal feed-forward neural networks, which cannot remember the past.

✅ Objective

To build a **stock price prediction model** using **RNN** on the **Google stock price dataset**.

📂 Dataset

- We used the dataset **Google_Stock_Price_Train.csv**, which contains daily stock price data.
- We selected only the **'Close'** column because it represents the final price of the stock each day and is commonly used for prediction.

📄 Code Explanation (Step-by-Step)

---

1. Importing Libraries

python

CopyEdit

import tensorflow as tf

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

- TensorFlow/Keras: To build the RNN model.
- Pandas: To load and handle the dataset.
- NumPy: For numerical operations.
- Matplotlib: For plotting results.

---

2. Loading the Dataset

python

CopyEdit

```
data = pd.read_csv('/content/sample_data/Google_Stock_Price_Train.csv')
data = data[["Close"]]
data.dropna(inplace=True)
```

- Only the 'Close' price is extracted.
- dropna() ensures any missing values are removed.

---

3. Data Preprocessing – Normalization

python

CopyEdit

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range=(0, 1))

data_scaled = scaler.fit_transform(data)
```

- Scaling the data to a 0-1 range is important so the model trains faster and avoids large error gradients.
- We use **MinMaxScaler** for this.

---

4. Creating Sequences for Time Series

python

CopyEdit

```
def create_sequences(data, sequence_length):
    X, Y = [], []
    for i in range(sequence_length, len(data)):
        X.append(data[i-sequence_length:i, 0])
        Y.append(data[i, 0])
    return np.array(X), np.array(Y)
```

- This function slides a window of sequence_length (e.g., 60 days) over the data.
- X holds 60 past values → input
- Y holds the next value → output

This helps the RNN learn a pattern between the past 60 days and the next day's price.

---

5. Reshape Input for RNN

python

CopyEdit

X = np.reshape(X, (X.shape[0], X.shape[1], 1))

- RNN expects 3D input: **[samples, time_steps, features]**.
- Since we only have one feature ('Close' price), the third dimension is 1.

---

6. Train-Test Split

python

CopyEdit

split = int(0.8 * len(X))

X_train, X_test = X[:split], X[split:]

Y_train, Y_test = Y[:split], Y[split:]

- We split 80% data for training and 20% for testing.

---

7. Build the RNN Model

python

CopyEdit

model = keras.Sequential([

```
    layers.SimpleRNN(50, return_sequences=True,
input_shape=(X_train.shape[1], 1)),
    layers.SimpleRNN(50),
    layers.Dense(1)
])
```

- **Sequential model**: A linear stack of layers.
- Two **SimpleRNN layers** with 50 neurons each:
  - First RNN layer returns sequences → needed if stacking multiple RNNs.
- Final **Dense layer** outputs the predicted stock price.

---

8. Compile the Model

python

CopyEdit

```
model.compile(optimizer='adam', loss='mean_squared_error')
```

- Optimizer = **Adam** (adaptive optimizer).
- Loss = **Mean Squared Error** (common for regression tasks like price prediction).

---

9. Train the Model

python

CopyEdit

history = model.fit(X_train, Y_train, epochs=50, batch_size=32, validation_split=0.1)

- We train the model for 50 epochs.
- Batch size = 32.
- 10% of training data used for validation.

---

10. Evaluate the Model

python

CopyEdit

loss = model.evaluate(X_test, Y_test)

print(f"Test Loss: {loss:.4f}")

- This gives us how well the model performs on unseen test data.

---

11. Make Predictions

python

CopyEdit

predicted_stock_price = model.predict(X_test)

- We predict stock prices for the test set.

---

12. Inverse Scale and Plot

python

CopyEdit

```
predicted_stock_price = scaler.inverse_transform(predicted_stock_price.reshape(-1, 1))

y_test_scaled = scaler.inverse_transform(Y_test.reshape(-1, 1))
```

- We reverse the scaling to get actual prices.

python

CopyEdit

```
plt.plot(y_test_scaled, color='blue', label='Actual Google Stock Price')

plt.plot(predicted_stock_price, color='red', label='Predicted Google Stock Price')

plt.title('Google Stock Price Prediction')

plt.xlabel('Time')

plt.ylabel('Stock Price')

plt.legend()

plt.show()
```

- Final graph shows **actual vs predicted stock prices**.

📘 Conclusion

We successfully built a **stock price prediction system** using **RNN** that learns from the past 60 days to predict the next day's price. The model was trained, tested, and gave us a visual comparison of predicted vs actual prices.

❓ Viva Questions with Answers

1. What are some variants of RNN?

- **LSTM (Long Short-Term Memory)** – solves vanishing gradient problem, remembers long-term.
- **GRU (Gated Recurrent Unit)** – simpler than LSTM, similar performance.
- **Bidirectional RNN** – reads sequences in both directions.
- **Deep RNN** – multiple RNN layers stacked.

2. What are the limitations of RNN?

- Cannot learn long-term dependencies (solved by LSTM/GRU).
- Training is slow due to sequential nature.
- Prone to vanishing or exploding gradients.
- Memory inefficient for very long sequences.