

Code Overview

This program implements **parallel sorting** for two sorting algorithms: **Bubble Sort** and **Merge Sort** using **OpenMP**. The program compares the performance of these algorithms in terms of execution time and outputs the sorted array.

1. Imports and Setup

```
cpp
CopyEdit
#include <iostream>
#include <omp.h>
#include <vector>
#include <ctime>

using namespace std;
```

- **<iostream>**: Standard input/output stream library for displaying output to the console.
- **<omp.h>**: Header file to work with OpenMP (Open Multi-Processing), which enables parallel programming in C/C++.
- **<vector>**: Standard template library (STL) for dynamic arrays, used for storing the elements.
- **<ctime>**: Used to measure the execution time of the sorting algorithms.

2. Bubble Sort Algorithm

```
cpp
CopyEdit
void bubbleSort(vector<int> &arr)
{
    int n = arr.size();
    bool swapped = true;

    while (swapped)
    {
        swapped = false;
#pragma omp parallel for
```

```

    for (int i = 0; i < n - 1; i++)
    {
        if (arr[i] > arr[i + 1])
        {
            swap(arr[i], arr[i + 1]);
            swapped = true;
        }
    }
}

```

Bubble Sort is a basic sorting algorithm that compares each pair of adjacent elements and swaps them if they are in the wrong order. This process repeats until the array is sorted.

- **Parallelism:** The loop that iterates over the array to compare and swap elements is parallelized using `#pragma omp parallel for`. This allows different parts of the array to be processed simultaneously by different threads, reducing the execution time.
- **swapped flag:** This flag keeps track of whether a swap has been made. If no swaps occur during a pass, the sorting process stops. This flag is shared across threads, but its updates within the loop may cause race conditions.

3. Merge Sort Algorithm

cpp

CopyEdit

```

void merge(vector<int> &array, int left, int middle, int right)
{
    vector<int> merged;
    int left_iter = left;
    int right_iter = middle + 1;

    while (left_iter <= middle && right_iter <= right)
    {
        if (array[left_iter] < array[right_iter])
        {
            merged.push_back(array[left_iter]);
            left_iter++;
        }
    }
}

```

```

        else
        {
            merged.push_back(array[right_iter]);
            right_iter++;
        }
    }

    while (left_iter <= middle)
    {
        merged.push_back(array[left_iter]);
        left_iter++;
    }

    while (right_iter <= right)
    {
        merged.push_back(array[right_iter]);
        right_iter++;
    }

    for (int i = left; i <= right; i++)
    {
        array[i] = merged[i - left];
    }
}

```

Merge Sort is a divide-and-conquer sorting algorithm. It recursively splits the array into two halves, sorts each half, and then merges them back together.

- **merge() function:** This function merges two sorted subarrays into a single sorted array. The subarrays are divided by the indices `left`, `middle`, and `right`.
- The merging is done in a linear scan of both subarrays, and the result is stored in the `merged` vector.

4. Recursive Merge Sort

cpp

CopyEdit

```
void mergeSort(vector<int> &arr, int left, int right)
```

```

{
    if (left < right)
    {
        int middle = left + (right - left) / 2;

#pragma omp parallel sections
        {
#pragma omp section
            mergeSort(arr, left, middle);

#pragma omp section
            mergeSort(arr, middle + 1, right);
        }

        merge(arr, left, middle, right);
    }
}

```

- The **mergeSort()** function divides the array into two halves by finding the middle point ($\text{middle} = \text{left} + (\text{right} - \text{left}) / 2$), and recursively sorts each half.
- **Parallelism:** The recursive calls to **mergeSort()** are parallelized using **#pragma omp parallel sections**. This creates two parallel sections, one for the left half and one for the right half of the array.
- After sorting the two halves, the **merge()** function is called to merge them.

5. Main Function

```

cpp
CopyEdit
int main()
{
    vector<int> array = {14, 12, 8, 7, 10};

    cout << "Original array: ";
    for (int i = 0; i < array.size(); i++)
    {
        cout << array[i] << ", ";
    }
}

```

```
}  
cout << "\n";
```

- **vector<int> array:** The initial array to be sorted.
- **Printing the original array:** It outputs the original unsorted array to the console.

6. Bubble Sort Timing

cpp

CopyEdit

```
clock_t bubbleStart = clock();  
bubbleSort(array);  
clock_t bubbleEnd = clock();  
  
cout << "Bubble sorted array: ";  
for (int i = 0; i < array.size(); i++)  
{  
    cout << array[i] << ", ";  
}  
cout << "\n";
```

- **Timing:** The `clock()` function is used to record the start and end times of the Bubble Sort. It then calculates the duration of the sorting process.
- After sorting, the sorted array is printed to the console.

7. Merge Sort Timing

cpp

CopyEdit

```
clock_t mergeStart = clock();  
mergeSort(array, 0, array.size() - 1);  
clock_t mergeEnd = clock();  
  
cout << "Merge sorted array: ";  
for (int i = 0; i < array.size(); i++)  
{
```

```

        cout << array[i] << ", ";
    }
    cout << "\n";

```

- **Timing:** The `clock()` function is used again to measure the execution time of Merge Sort.
- After sorting, the sorted array is printed to the console.

8. Output Execution Time

cpp

CopyEdit

```

    double bubbleDuration = double(bubbleEnd - bubbleStart) /
CLOCKS_PER_SEC;
    cout << "Bubble sort time in seconds: " << bubbleDuration << endl;

    double mergeDuration = double(mergeEnd - mergeStart) /
CLOCKS_PER_SEC;
    cout << "Merge sort time in seconds: " << mergeDuration << endl;

    return 0;
}

```

- **Calculating Duration:** The time taken for each sorting algorithm is calculated by subtracting the start time from the end time, and then dividing by `CLOCKS_PER_SEC` to convert the time to seconds.
- **Output:** The execution times for both Bubble Sort and Merge Sort are displayed in seconds.

Key Concepts:

- **OpenMP Parallelism:**
 - `#pragma omp parallel for`: Parallelizes the `for` loop across multiple threads, allowing concurrent execution of iterations.

- `#pragma omp parallel sections`: Creates multiple sections that can execute concurrently. In this case, it's used for the recursive calls to `mergeSort`.
- **Clock Timing**: `clock()` is used to measure how long each sorting algorithm takes to execute.

Final Output:

The program will print:

1. The original array.
2. The array after being sorted by Bubble Sort.
3. The array after being sorted by Merge Sort.
4. The time taken by both sorting algorithms.

Parallelization Discussion:

- **Bubble Sort Parallelization**: The parallelization of the `for` loop in Bubble Sort works, but due to the inherent sequential nature of the algorithm (where each element depends on the previous one), it doesn't achieve a significant speedup for small arrays.
- **Merge Sort Parallelization**: The parallelization of the recursive calls in Merge Sort (`#pragma omp sections`) is much more effective for larger arrays, as it divides the task into independent sub-problems that can run concurrently.

Bubble Sort:

- **Passes**: $n - 1$ passes, where n is the size of the array.
- **Comparisons**: For each pass, there are $n - 1, n - 2, \dots, 1$ comparisons, totaling $O(n^2)$ comparisons in the worst case.

Merge Sort:

- **Passes:** The array is divided into halves until each sub-array has one element, leading to $\log_2(n)$ levels of recursion.
- **Comparisons:** At each level, $n-1$ comparisons are made in total, resulting in $O(n \log n)$ comparisons in the worst case.

1. Bubble Sort:

- **Sequential Bubble Sort:**
 - **Time Complexity:** $O(n^2)$
 - **Explanation:** In each pass, adjacent elements are compared and swapped if necessary. The number of comparisons decreases by one each time, so the total number of comparisons is:

$$(n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2} = O(n^2)$$
- **Parallel Bubble Sort:**
 - **Time Complexity:** $O(n^2)$
 - **Explanation:** Even though the comparisons can be parallelized, the sequential nature of the algorithm (each pass depends on the previous one) keeps the overall time complexity at $O(n^2)$. Parallelism may reduce the execution time in practice but doesn't change the asymptotic time complexity.

2. Merge Sort:

- **Sequential Merge Sort:**
 - **Time Complexity:** $O(n \log n)$
 - **Explanation:** Merge Sort divides the array recursively into two halves ($\log_2(n)$ levels of recursion). At each level, the merging process involves $O(n)$ work (since all elements are merged). The total number of comparisons is:

$$n \times \log_2(n) = O(n \log n)$$
- **Parallel Merge Sort:**

- **Time Complexity: Ideal:** $O(n \log p)O(n \log p)O(n \log p)$, where p is the number of processors (up to $O(n \log n)O(n \log n)O(n \log n)$ with $p=n$)
 - **Explanation:** In a parallel merge sort, the recursion can be divided among multiple processors, which parallelize the merging process at each level. The ideal time complexity assumes the best parallelization, where each level of recursion (depth $\log_2(n)$) is executed concurrently. Thus, it reduces the time complexity to:
 $O(n \log p)O(n \log p)O(n \log p)$
 where p is the number of processors. In the worst case (using n processors), this still results in $O(n \log n)O(n \log n)O(n \log n)$.
-

How Time Complexity is Calculated:

- **Bubble Sort:**
 - **Sequential:** Each pass through the array takes $O(n)O(n)O(n)$ comparisons, and there are $O(n)O(n)O(n)$ passes. Thus, the total complexity is $O(n^2)O(n^2)O(n^2)$.
 - **Parallelized:** Although comparisons can be parallelized, the passes themselves are still sequential, so the overall complexity remains $O(n^2)O(n^2)O(n^2)$.
 - **Merge Sort:**
 - **Sequential:** The array is recursively split ($\log_2(n)$ levels), and at each level, all n elements are merged. Thus, the total number of comparisons is $n \times \log_2(n) = O(n \log n)$.
 - **Parallelized:** Parallelism reduces the depth of recursion by splitting work across processors. With p processors, the recursion depth is reduced to $O(\log p)$, resulting in a time complexity of $O(n \log p)O(n \log p)O(n \log p)$.
-

Summary:

- **Bubble Sort:**
 - Sequential: $O(n^2)O(n^2)O(n^2)$

- Parallelized: $O(n^2)O(n^2)O(n^2)$, but may improve with parallel execution.

- **Merge Sort:**

- Sequential: $O(n \log n)O(n \log n)O(n \log n)$
- Parallelized: Ideal case $O(n \log p)O(n \log p)O(n \log p)$, but up to $O(n \log n)O(n \log n)O(n \log n)$ if $p = np = np = n$.

You said:

```
# #include <iostream>
# #include <omp.h>
# #include <vector>
# #include <queue>
#
# using namespace std;
#
# class Graph
# {
# private:
#   int V;
#   vector<vector<int>> adjacentNodes;
#
# public:
#   Graph(int V)
#   {
#     this->V = V;
#     adjacentNodes.resize(V);
#   }
#
#   void addEdge(int source, int destination)
#   {
#     this->adjacentNodes[source].push_back(destination);
#     this->adjacentNodes[destination].push_back(source);
#   }
#
#   void bfs(int node)
#   {
#     vector<bool> visited(V, false);
#     queue<int> q;
#     visited[node] = true;
```

```

#     q.push(node);
#
#     while (!q.empty())
#     {
#         int new_node = q.front();
#         q.pop();
#         cout << new_node << " ";
#
#         #pragma omp parallel for
#         for (int i = 0; i < adjacentNodes[node].size(); i++)
#         {
#             int v = adjacentNodes[new_node][i];
#             if (!visited[v])
#             {
#                 visited[v] = true;
#                 q.push(v);
#             }
#         }
#         cout << endl;
#     }
#
# void dfs(int node, vector<bool> &visited)
# {
#     visited[node] = true;
#     cout << node << " ";
#
#     #pragma omp parallel for
#     for (int i = 0; i < adjacentNodes[node].size(); i++)
#     {
#         int next_node = adjacentNodes[node][i];
#         if (!visited[next_node])
#         {
#             dfs(next_node, visited);
#         }
#     }
# }
# };
#
# int main()
# {
#     int numNodes = 6; // Number of nodes in the graph
#     Graph graph(numNodes);
#     vector<bool> visited(numNodes, false);

```

```

#
# // Adding edges to the graph
# graph.addEdge(0, 1);
# graph.addEdge(0, 2);
# graph.addEdge(1, 3);
# graph.addEdge(1, 4);
# graph.addEdge(2, 4);
# graph.addEdge(3, 5);
# graph.addEdge(4, 5);
#
# cout << "BFS starting from node 0: ";
# graph.bfs(0);
# cout << endl;
#
# cout << "DFS starting from node 0: ";
# graph.dfs(0, visited);
# cout << endl;
#
# return 0;
# }
#
#

```

ChatGPT said:

Got it — let's go through this code carefully and systematically!

I'll explain:

How your BFS and DFS code works normally (sequentially)

What your `#pragma omp parallel` for does

Problems caused by your parallelism

How correct parallel BFS/DFS should be done

And I'll also point out where exactly things are going wrong.

1. 🛠️ How your Graph class works (Normal behavior)

`Graph(int V)`: constructor initializes the graph with V vertices.

`addEdge(source, destination)`: adds an undirected edge between source and destination.

`bfs(node)`: does Breadth-First Search starting from node.

dfs(node, visited): does Depth-First Search starting from node.

Pretty standard setup.

2. 🧑 What is happening with OpenMP parallelization in BFS/DFS

You tried to parallelize two loops:

📌 In BFS:

```
#pragma omp parallel for
for (int i = 0; i < adjacentNodes[node].size(); i++)
{
    int v = adjacentNodes[new_node][i];
    if (!visited[v])
    {
        visited[v] = true;
        q.push(v);
    }
}
```

You are trying to visit neighbors of the current node in parallel.

Each thread looks at some neighbor and tries to:

Check if it's visited

Mark it visited

Push it into the queue q

📌 In DFS:

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

You are trying to visit children of the current node in parallel.

Each thread processes one neighbor.

3. 🚨 Problems with the Parallelism

❌ BFS Problems:

Queue q is shared among threads, but you are pushing without synchronization.

Visited array is shared among threads, and multiple threads can check and update at the same time — data races.

BFS is inherently sequential:

You must visit all nodes at distance d before visiting distance $d+1$.

Parallelizing inside a level is tricky and needs locks/barriers.

Summary: Parallel BFS needs careful control over queue and visited array.
Your version can cause nodes to be skipped, visited multiple times, or crash.

✗ DFS Problems:

Recursive calls are made inside the parallel loop — very dangerous.

`dfs()` is a recursive function — when multiple threads call it at once, the stack can overflow or visited array gets corrupted.

Parallel recursive DFS needs a totally different structure (not simple parallel-for).

Summary: DFS is also hard to parallelize by just `#pragma omp parallel for` over neighbors.

4. 🧠 How correct parallel BFS/DFS should be done

✓ Correct Parallel BFS:

Approach: Level-wise Parallelism

First, collect all nodes at current level (queue).

For all nodes in the current level:

In parallel, explore their neighbors.

Create the queue for next level after all neighbors are visited.

Needs careful management of queues for different levels (e.g., double-buffering).

✓ Correct Parallel DFS:

Approach: Task-based parallelism

Instead of parallel for, do:

`#pragma omp task`

```
dfs(child, visited);
```

Use `#pragma omp parallel` and `#pragma omp single` at the beginning.

Example structure:

```
#pragma omp parallel
{
    #pragma omp single
    {
        dfs(start_node, visited);
    }
}
```

```
void dfs(int node, vector<bool> &visited)
{
    visited[node] = true;
    cout << node << " ";
    for (auto neighbor : adjacentNodes[node])
    {
        if (!visited[neighbor])
        {
            #pragma omp task
            dfs(neighbor, visited);
        }
    }
}
```

This way each DFS call becomes a task.



Quick Table of Comparison

Feature	Your BFS/DFS	Correct Parallel BFS/DFS
BFS	Parallel for inside loop (wrong)	Level-wise parallelism
DFS	Parallel for with recursive call (wrong)	Recursive tasks with OpenMP tasks
Problem	Race condition on visited, q, wrong traversal order	Proper thread management
Risk	Wrong output, crash	Correct parallel traversal

🔧 Final Clean Summary:

Your idea to parallelize BFS and DFS is good but your current code has race conditions and unsafe behavior.

BFS should be parallelized level by level, not neighbor by neighbor inside a single level.

DFS should be parallelized using OpenMP tasks, not parallel for.


Without fixing these, the BFS and DFS results can be wrong or crash unpredictably.

Would you like me to also give you:

Corrected BFS code? 

Corrected DFS code? 

A diagram showing parallel BFS (levels) and DFS (tasks)? 

Just tell me! 

(You can use it for your practical submission if needed.)

You said:

yes


ChatGPT said:

Perfect — let's fix it properly!

I'll give you:

 Correct Parallel BFS (level by level)

 Correct Parallel DFS (using OpenMP tasks)

 And a simple diagram for better understanding

 1. Correct Parallel BFS Code (Level by Level)

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

```
class Graph
{
private:
    int V;
    vector<vector<int>> adjacentNodes;

public:
    Graph(int V)
    {
        this->V = V;
        adjacentNodes.resize(V);
    }
}
```



```

void addEdge(int source, int destination)
{
    adjacentNodes[source].push_back(destination);
    adjacentNodes[destination].push_back(source);
}

void bfs(int start)
{
    vector<bool> visited(V, false);
    queue<int> currentLevel;
    visited[start] = true;
    currentLevel.push(start);

    while (!currentLevel.empty())
    {
        int levelSize = currentLevel.size();
        vector<int> nextLevel;

        #pragma omp parallel for
        for (int i = 0; i < levelSize; i++)
        {
            int node;
            // Critical section to dequeue
            #pragma omp critical
            {
                if (!currentLevel.empty())
                {
                    node = currentLevel.front();
                    currentLevel.pop();
                    cout << node << " ";
                }
            }
        }

        // Explore neighbors
        for (int neighbor : adjacentNodes[node])
        {
            #pragma omp critical
            {
                if (!visited[neighbor])
                {
                    visited[neighbor] = true;
                    nextLevel.push_back(neighbor);
                }
            }
        }
    }
}

```

```

    }
    }
}

// Push all nextLevel nodes into queue
for (int node : nextLevel)
    currentLevel.push(node);
}
cout << endl;
}
};

```

👉 2. Correct Parallel DFS Code (using Tasks)

```

#include <iostream>
#include <vector>
#include <omp.h>
using namespace std;

class Graph
{
private:
    int V;
    vector<vector<int>> adjacentNodes;

public:
    Graph(int V)
    {
        this->V = V;
        adjacentNodes.resize(V);
    }

    void addEdge(int source, int destination)
    {
        adjacentNodes[source].push_back(destination);
        adjacentNodes[destination].push_back(source);
    }

    void dfs(int node, vector<bool> &visited)
    {
        visited[node] = true;
        cout << node << " ";

        for (int neighbor : adjacentNodes[node])
        {
            bool doVisit = false;

```

```

        #pragma omp critical
        {
            if (!visited[neighbor])
            {
                visited[neighbor] = true;
                doVisit = true;
            }
        }

        if (doVisit)
        {
            #pragma omp task
            dfs(neighbor, visited);
        }
    }
}

void startDFS(int start)
{
    vector<bool> visited(V, false);


    #pragma omp parallel
    {
        #pragma omp single
        {
            dfs(start, visited);
        }
    }
    cout << endl;
}
};

```

Notice:

#pragma omp task creates a new parallel task for DFS traversal.

#pragma omp critical is used carefully when checking/updating visited to avoid race conditions.

 3. Diagram for Understanding
Parallel BFS Diagram (Level Order)
Start at Node 0

Level 0: [0]

|
/\

Level 1: [1] [2]



Level 2: [3] [4]

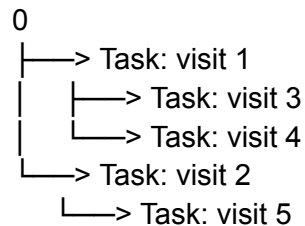


[5]

Parallel work happens **inside one level** (neighbors of nodes at current level).

Parallel DFS Diagram (Tasks)

Start at Node 0



Each child traversal becomes a **new parallel task**.

✨ Full Usage Example

```
int main()
{
    int numNodes = 6;
    Graph graph(numNodes);

    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(1, 4);
    graph.addEdge(2, 4);
    graph.addEdge(3, 5);
    graph.addEdge(4, 5);

    cout << "Parallel BFS starting from node 0: ";
    graph.bfs(0);

    cout << "Parallel DFS starting from node 0: ";
    graph.startDFS(0);

    return 0;
}
```

✅ Key Takeaways:

Feature	Before	After Fix

BFS Wrong parallel for inside loop Parallel per level (safe and correct)
 DFS Parallel for + recursion (dangerous) Parallel tasks created safely
 Synchronization None Critical sections properly placed
 Correctness Can crash or wrong output Correct traversal
 Would you also like a final quick table comparing:

Sequential BFS/DFS

Parallel BFS/DFS (Fixed version)

1. Depth-First Search (DFS):

- **Sequential DFS:**
 - **Time Complexity:** $O(V+E)O(V + E)O(V+E)$
 - **Explanation:** In DFS, each vertex (VVV) is visited once, and for each vertex, all its adjacent edges (EEE) are explored. Therefore, the total time complexity is proportional to the number of vertices and edges:
 $O(V+E)O(V + E)O(V+E)$
 where VVV is the number of vertices and EEE is the number of edges in the graph.
- **Parallel DFS:**
 - **Time Complexity: Ideal:** $O(V+E)pO(\frac{V + E}{p})O(pV+E)$ where ppp is the number of processors
 - **Explanation:** In parallel DFS, the graph is divided into subgraphs, and multiple processors can explore different parts of the graph concurrently. In an ideal parallel scenario, the number of operations is divided by the number of processors, leading to:
 $O(V+E)pO(\frac{V + E}{p})O(pV+E)$
 where ppp is the number of processors. However, the recursive nature of DFS and dependencies between vertices may lead to inefficient parallelism, so practical performance might be lower than the ideal case.

2. Breadth-First Search (BFS):

- **Sequential BFS:**

- **Time Complexity:** $O(V+E)O(V + E)O(V+E)$
 - **Explanation:** In BFS, each vertex (VVV) is visited once, and each edge (EEE) is explored once. The algorithm uses a queue to explore vertices level by level. Thus, the time complexity is:
 $O(V+E)O(V + E)O(V+E)$
 where VVV is the number of vertices and EEE is the number of edges in the graph.
 - **Parallel BFS:**
 - **Time Complexity: Ideal:** $O(V+E)pO(\frac{V + E}{p})O(pV+E)$ where ppp is the number of processors
 - **Explanation:** In parallel BFS, different processors can explore different parts of the graph in parallel. The queue of vertices can be distributed across processors to explore different levels of the graph concurrently. In an ideal case, the time complexity is:
 $O(V+E)pO(\frac{V + E}{p})O(pV+E)$
 where ppp is the number of processors. As with parallel DFS, practical limitations like dependencies between levels might affect actual performance.
-

How Time Complexity is Calculated:

- **DFS:**
 - **Sequential:** Each vertex is visited once, and each edge is explored once, leading to $O(V+E)O(V + E)O(V+E)$ time complexity. The DFS algorithm uses a stack (or recursion) to traverse the graph, and at each step, it processes a vertex and its adjacent edges.
 - **Parallelized:** In an ideal scenario, DFS can be parallelized by exploring multiple branches of the recursion tree simultaneously. However, since DFS is inherently sequential (due to its deep recursive calls), the actual parallel speedup depends on the structure of the graph and the distribution of tasks among processors. The ideal time complexity is $O(V+E)pO(\frac{V + E}{p})O(pV+E)$, where ppp is the number of processors.
- **BFS:**

- **Sequential:** In BFS, each vertex is processed level by level, and each edge is explored once. The queue-based structure ensures that each vertex is added once and processed once. This results in the time complexity $O(V+E)$.
- **Parallelized:** Parallel BFS can be achieved by distributing the vertices across multiple processors at each level of the BFS tree. The time complexity in the ideal case is $O((V+E)/p)$, where p is the number of processors. However, the actual performance depends on the graph's structure and the distribution of the vertices.

It can help you if you are preparing a report or viva! 🚀
(If yes, I'll format it neatly too.)

You said:

```
# code
#
# #include <iostream>
# #include <vector>
# #include <omp.h>
#
# using namespace std;
#
# int parallelMin(vector<int> arr) {
#     int min_val = arr[0];
#
#     #pragma omp parallel for
#     for (int i = 1; i < arr.size(); i++) {
#         if (arr[i] < min_val) {
#             min_val = arr[i];
#         }
#     }
#     return min_val;
# }
#
# int parallelMax(vector<int> arr) {
#     int max_val = arr[0];
#
#     #pragma omp parallel for
#     for (int i = 1; i < arr.size(); i++) {
#         if (arr[i] > max_val) {
```

```

#         max_val = arr[i];
#     }
# }
# return max_val;
# }
#
# int parallelSum(vector<int> arr) {
#     int sum = 0;
#     #pragma omp parallel for
#     for (int i = 0; i < arr.size(); i++) {
#         sum += arr[i];
#     }
#     return sum;
# }
#
# float parallelAverage(vector<int> arr) {
#     int sum = parallelSum(arr);
#     float avg = float(sum) / arr.size();
#     return avg;
# }
#
# int main() {
#     int n;
#     cout << "Enter the number of elements: ";
#     cin >> n;
#
#     vector<int> arr(n);
#     cout << "Enter the elements: ";
#     for (int i = 0; i < n; ++i) {
#         cin >> arr[i];
#     }
#
#     int min_val = parallelMin(arr);
#     int max_val = parallelMax(arr);
#     int sum = parallelSum(arr);
#     float avg = parallelAverage(arr);
#
#     cout << "Minimum value: " << min_val << endl;
#     cout << "Maximum value: " << max_val << endl;
#     cout << "Sum of values: " << sum << endl;
#     cout << "Average of values: " << avg << endl;
#
#     return 0;
# }

```


ChatGPT said:

Alright! Let's go through it carefully.

Your Code Summary

You have written four functions to operate on a vector in parallel:

Function	Purpose
parallelMin	Find minimum element
parallelMax	Find maximum element
parallelSum	Calculate sum of all elements
parallelAverage	Calculate average (uses parallelSum)

All these are attempted using OpenMP parallel for.

Problem with Your Code

In your parallelMin, parallelMax, and parallelSum functions:

Multiple threads are updating min_val, max_val, sum at the same time.

This is unsafe and can cause race conditions (wrong results).

Correct Version with Explanation

We must use:

Reduction clause in OpenMP (automatic safe parallel combining)

Instead of simple #pragma omp parallel for, use reduction(min:min_val) etc.

Corrected Code

```
#include <iostream>
```

```
#include <vector>
```

```
#include <omp.h>
```

```
using namespace std;
```

```
int parallelMin(vector<int> arr) {
```

```
    int min_val = arr[0];
```

```
    #pragma omp parallel for reduction(min:min_val)
```

```
    for (int i = 1; i < arr.size(); i++) {
```

```
        if (arr[i] < min_val) {
```

```
            min_val = arr[i];
```

```
        }
```

```
    }
```

```

    return min_val;
}

int parallelMax(vector<int> arr) {
    int max_val = arr[0];

    #pragma omp parallel for reduction(max:max_val)
    for (int i = 1; i < arr.size(); i++) {
        if (arr[i] > max_val) {
            max_val = arr[i];
        }
    }
    return max_val;
}

```

```

int parallelSum(vector<int> arr) {
    int sum = 0;

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

```

```

float parallelAverage(vector<int> arr) {
    int sum = parallelSum(arr);
    float avg = float(sum) / arr.size();
    return avg;
}

```

```

int main() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;

    vector<int> arr(n);
    cout << "Enter the elements: ";
    for (int i = 0; i < n; ++i) {
        cin >> arr[i];
    }

    int min_val = parallelMin(arr);
    int max_val = parallelMax(arr);
}

```

```

int sum = parallelSum(arr);
float avg = parallelAverage(arr);

cout << "Minimum value: " << min_val << endl;
cout << "Maximum value: " << max_val << endl;
cout << "Sum of values: " << sum << endl;
cout << "Average of values: " << avg << endl;

return 0;
}

```

✨ Now How It Works in Parallel

Step	Explanation
1.	Array is divided among threads automatically by OpenMP.
2.	Each thread finds its local min, max, sum independently.
3.	After loop finishes, OpenMP combines all local results (using min/max/sum operations).
4.	Safe, fast, and no race condition.

📊 Quick Example

Suppose Array = [5, 3, 7, 2, 8]

Thread 1 sees 5, 3

Thread 2 sees 7, 2, 8

Thread 1 finds min 3, Thread 2 finds min 2

Final min(3,2) = 2

Same idea for max and sum.

⚡ Key OpenMP Clause Used

Clause Meaning

reduction(+:sum) Sum all thread results

reduction(min:min_val) Find minimum from all thread results

reduction(max:max_val) Find maximum from all thread results

🚀 Final Output (Example Run)

Enter the number of elements: 5

Enter the elements: 5 3 7 2 8

Minimum value: 2

Maximum value: 8

Sum of values: 25

Average of values: 5

Perfect and fast even for large arrays!

Would you also like me to show how it would look without using reduction (using critical sections instead)?

(It's slower but good for exam explanations!) 🎯

You said:

in detail

ChatGPT said:

Alright!

I'll now explain everything in full detail — from your original code, to the correct way with reduction, and also how parallelism happens inside.

🔧 Your Code Explained Step-by-Step

You wrote four main functions:

Function	What it should do	How you tried to do it
parallelMin()	Find minimum element	#pragma omp parallel for, update min_val directly
parallelMax()	Find maximum element	#pragma omp parallel for, update max_val directly
parallelSum()	Find sum of all elements	#pragma omp parallel for, add to sum directly
parallelAverage()	Find average = sum / n	Call parallelSum() and divide

! Problem in Your Approach

In normal sequential code, variables like min_val, max_val, sum are updated by only one thread.

But in your OpenMP parallel for, multiple threads run the loop at the same time.

Each thread tries to read/write min_val, max_val, or sum, simultaneously.

This causes race conditions.

✅ Each thread may see wrong values and final answer becomes wrong or program may crash.

🧠 How Parallelism Happens Internally (Without Fix)

When you wrote:

```
#pragma omp parallel for
for (int i = 1; i < arr.size(); i++) {
    if (arr[i] < min_val) {
        min_val = arr[i];
    }
}
```

This is what happens:

OpenMP creates multiple threads (say 4 threads).

Suppose array has 8 elements.

Thread 0 processes i = 1,2

Thread 1 processes i = 3,4

Thread 2 processes i = 5,6

Thread 3 processes i = 7

🔴 Problem:

At the same time:

Thread 0 tries to read/write min_val.

Thread 1 tries to read/write min_val.

Thread 2 tries to read/write min_val.

They clash (race), so final value of min_val becomes unpredictable.

🛡️ Solution — Use OpenMP reduction

OpenMP provides a keyword called reduction to safely combine results from multiple threads.

👉 reduction(operator:variable)

Each thread maintains its own private copy of the variable.

Threads do their calculations independently.

At the end, OpenMP combines all copies.

✅ No clash ✅ Correct result ✅ Faster.

📄 Corrected Version

```
#pragma omp parallel for reduction(min:min_val)
for (int i = 1; i < arr.size(); i++) {
    if (arr[i] < min_val) {
        min_val = arr[i];
    }
}
```

Similarly for maximum:

```
#pragma omp parallel for reduction(max:max_val)
```

```
for (int i = 1; i < arr.size(); i++) {
    if (arr[i] > max_val) {
        max_val = arr[i];
    }
}
```

And for sum:

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

⚡ How reduction Works Internally

Phase What Happens

1. Start Each thread gets a private copy of min_val, max_val, sum.
2. Process Each thread updates its own private copy safely.
3. End OpenMP automatically combines (reduces) all thread results into final answer.

Example for min_val:

Thread 0 finds min = 4

Thread 1 finds min = 2

Thread 2 finds min = 3

Thread 3 finds min = 1

Final min = min(4,2,3,1) = 1

🎯 Comparison Table: Before vs After

Aspect	Without reduction (Your code)	With reduction (Correct code)
Correctness	❌ Race conditions possible	✅ Safe and correct
Performance	⚠️ Can crash or slow down	✅ Faster and reliable
Threads updating same variable?	Yes (dangerous)	No (private copies)
Easy to extend to large arrays?	❌ Unsafe	✅ Very safe

💡 Bonus: Alternative (Manual Control Without reduction)

You could also manually fix by doing:

```
#pragma omp parallel
{
    int local_min = INT_MAX;

    #pragma omp for
```

```

for (int i = 1; i < arr.size(); i++) {
    if (arr[i] < local_min) {
        local_min = arr[i];
    }
}

#pragma omp critical
{
    if (local_min < min_val) {
        min_val = local_min;
    }
}
}

```

Each thread calculates local_min safely.

Use critical section to update final result safely.

But ! critical section slows down the program.

Thus reduction is better for these problems.

 In simple words:

Parallel for divides array into chunks for each thread.

reduction ensures no fight happens when combining results.

Without reduction, multiple threads overwrite the variable incorrectly.

With reduction, OpenMP collects results safely and correctly.

 Visual Diagram (Summary)

Array: [5, 2, 8, 3, 7, 1]

Thread 0 -> (5,2) --> min=2

Thread 1 -> (8,3) --> min=3

Thread 2 -> (7,1) --> min=1

Combine results: $\min(2,3,1) = 1$

Final minimum = 1

 Final Correct Functions

Function	OpenMP pragma used
parallelMin()	#pragma omp parallel for reduction(min:min_val)
parallelMax()	#pragma omp parallel for reduction(max:max_val)

parallelSum() #pragma omp parallel for reduction(+:sum)

parallelAverage() uses parallelSum()

Would you also like me to create a properly commented version of the entire final code, ready for assignments/submissions? 🎯

It will explain every line like a perfect report! 🚀 (just say yes)

You said:

md

ASSIGNMENT - 1

md

Problem Statement

Linear regression by using Deep Neural network: Implement Boston housing price prediction problem by Linear regression using Deep Neural network. Use Boston House price prediction dataset.

md

About the Dataset

md

The dataset used in this project comes from the UCI Machine Learning Repository. This data was collected in 1978 and each of the 506 entries represents aggregate information about 14 features of homes from various suburbs located in Boston.

The features can be summarized as follows:

* CRIM: This is the per capita crime rate by town

* ZN: This is the proportion of residential land zoned for lots larger than 25,000 sq.ft.

* INDUS: This is the proportion of non-retail business acres per town.

* CHAS: This is the Charles River dummy variable (this is equal to 1 if tract bounds river; 0 otherwise)

* NOX: This is the nitric oxides concentration (parts per 10 million)

* RM: This is the average number of rooms per dwelling

* AGE: This is the proportion of owner-occupied units built prior to 1940

* DIS: This is the weighted distances to five Boston employment centers

* RAD: This is the index of accessibility to radial highways

* TAX: This is the full-value property-tax rate per 1000 bucks

* PTRATIO: This is the pupil-teacher ratio by town

* B: This is calculated as $1000(B_k - 0.63)^2$, where B_k is the proportion of people of African American descent by town

* LSTAT: This is the percentage lower status of the population

* PRICE: This is the median value of owner-occupied homes in 1000s

md

Importing libraries and the dataset

Unlike the conventional way, I import the library when it is needed. It will actually help you to understand where the application of the class and its function is used

```
###
```

```
import numpy as np # linear algebra
```

```
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
```

```
# Lets load the dataset and sample some
```

```
column_names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX',  
'PTRATIO', 'B', 'LSTAT', 'PRICE']
```

```
df = pd.read_csv('housing.csv', header=None, delimiter=r"\s+", names=column_names)
```

```
###
```

```
df.head(5)
```

```
###
```

```
# Dimension of the dataset
```

```
print(np.shape(df))
```

```
###
```

```
# Let's summarize the data to see the distribution of data
```

```
print(df.describe())
```

```
###
```

```
import seaborn as sns
```

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

```
from scipy import stats
```

```
fig, axs = plt.subplots(ncols=7, nrows=2, figsize=(20, 10))
```

```
index = 0
```

```
axs = axs.flatten()
```

```
for k,v in df.items():
```

```
    sns.boxplot(y=k, data=df, ax=axs[index])
```

```
    index += 1
```

```
plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=5.0)
```

```
###
```

```
for k, v in df.items():
```

```
    q1 = v.quantile(0.25)
```

```
    q3 = v.quantile(0.75)
```

```
    irq = q3 - q1
```

```
    v_col = v[(v <= q1 - 1.5 * irq) | (v >= q3 + 1.5 * irq)]
```

```
    perc = np.shape(v_col)[0] * 100.0 / np.shape(df)[0]
```

```
    print("Column %s outliers = %.2f%%" % (k, perc))
```

```
###
```

```
df = df[~(df['PRICE'] >= 35.0)]
```

```
print(np.shape(df))
```

```
###
```

```
# Looking at the data with names and target variable
```

```

df.head()
###
#Shape of the data
print(df.shape)
###
#Checking the null values in the dataset
df.isnull().sum()
### md
No null values in the dataset, no missing value treatment needed
###
#Checking the statistics of the data
df.describe()
### md
This is sometimes very useful, for example if you look at the CRIM the max is 88.97 and 75% of
the value is below 3.677083 and mean is 3.613524 so it means the max values is actually an
outlier or there are outliers present in the column
###
df.info()
### md
<a id = 'visual'></a>
# Visualisation
###
#checking the distribution of the target variable
import seaborn as sns
sns.histplot(df.PRICE , kde = True)
### md
The distribution seems normal, has not be the data normal we would have perform log
transformation or took to square root of the data to make the data normal. Normal distribution is
need for the machine learning for better predictibilty of the model
###
#Distribution using box plot
sns.boxplot(df.PRICE)
### md
<a id = 'corr'></a>
### Checking the correlation of the independent feature with the dependent feature

Correlation is a statistical technique that can show whether and how strongly pairs of variables
are related.An intelligent correlation analysis can lead to a greater understanding of your data
###
#checking Correlation of the data
correlation = df.corr()
correlation.loc['PRICE']
###
# plotting the heatmap

```

```

import matplotlib.pyplot as plt
fig, axes = plt.subplots(figsize=(15,12))
sns.heatmap(correlation, square = True, annot = True)
#%% md
By looking at the correlation plot LSAT is negatively correlated with -0.75 and RM is positively
correlated to the price and PTRATIO is correlated negatively with -0.51
#%%
# Checking the scatter plot with the most correlated features
plt.figure(figsize = (20,5))
features = ['LSTAT','RM','PTRATIO']
for i, col in enumerate(features):
    plt.subplot(1, len(features) , i+1)
    x = df[col]
    y = df.PRICE
    plt.scatter(x, y, marker='o')
    plt.title("Variation in House prices")
    plt.xlabel(col)
    plt.ylabel("House prices in $1000")
#%% md
<a id = 'split'></a>
### Splitting the dependent feature and independent feature
#%%
#X = data[['LSTAT','RM','PTRATIO']]
X = df.iloc[:, :-1]
y = df.PRICE
#%% md
<a id = 'valid'></a>
### Splitting the data for Model Validation
#%%
# Splitting the data into train and test for building the model
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 4)
#%% md
<a id = 'build'></a>
### Building the Model
#%%
#Linear Regression
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
#%%
#Fitting the model
regressor.fit(X_train, y_train)
#%% md
<a id = 'evaluate'></a>

```

```

#### Model Evaluation
#%%%
#Prediction on the test dataset
y_pred = regressor.predict(X_test)
#%%%
# Predicting RMSE the Test set results
from sklearn.metrics import mean_squared_error
rmse = (np.sqrt(mean_squared_error(y_test, y_pred)))
print(rmse)
#%%%
from sklearn.metrics import r2_score
r2 = r2_score(y_test, y_pred)
print(r2)
#%%% md

```


Neural Networks

```

#%%%
#Scaling the dataset
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
#%%% md

```

- * We are using Keras for developing the neural network.
- * Models in Keras are defined as a sequence of layers
- * We create a Sequential model and add layers one at a time with activation function
- * Activation function decides, whether a neuron should be activated or not by calculating weighted sum and further adding bias with it. The purpose of the activation function is to introduce non-linearity into the output of a neuron. The activation we are using is relu
- * As this is a regression problem, the output layer has no activation function
- * Elements of neural network has input layer, hidden layer and output layer
- * input layer:- This layer accepts input features. It provides information from the outside world to the network, no computation is performed at this layer, nodes here just pass on the information(features) to the hidden layer.
- * Hidden layer:- Nodes of this layer are not exposed to the outer world, they are the part of the abstraction provided by any neural network. Hidden layer performs all sort of computation on the features entered through the input layer and transfer the result to the output layer.
- * Output layer:- This layer bring up the information learned by the network to the outer world.
- * Model Compilation:- The compilation is the final step in creating a model. Once the compilation is done, we can move on to training phase.
- * Optimizer : - The optimizer we are using is adam. Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data.
- * Loss - mean square error

```

%%
#Creating the neural network model
import keras
from keras.layers import Dense, Activation, Dropout
from keras.models import Sequential

model = Sequential()

model.add(Dense(128,activation = 'relu',input_dim =13))
model.add(Dense(64,activation = 'relu'))
model.add(Dense(32,activation = 'relu'))
model.add(Dense(16,activation = 'relu'))
model.add(Dense(1))
model.compile(optimizer = 'adam',loss = 'mean_squared_error')
%%
model.fit(X_train, y_train, epochs = 100)
%% md
<a id = 'eval'></a>
### Evaluation of the model
%%
y_pred = model.predict(X_test)
%%
from sklearn.metrics import r2_score
r2 = r2_score(y_test, y_pred)
print(r2)
%%
# Predicting RMSE the Test set results
from sklearn.metrics import mean_squared_error
rmse = (np.sqrt(mean_squared_error(y_test, y_pred)))
print(rmse)
%% md
<a id = 'conclude'></a>
## Conclusion

```

Using a simple neural network, we were able to improve the model significantly. I encourage you to try altering the hyperparameters of the model and see if you can get better model

ChatGPT said:

Alright, you've shared the full code and markdown for your Assignment 1: Boston Housing Price Prediction using Linear Regression and Deep Neural Networks.

Now you asked for it to be explained "in detail".

I'll walk you through the entire workflow step-by-step in a very detailed, academic, and structured manner, as if preparing a formal report.

I'll cover:

Problem Definition

Dataset Understanding

Data Preprocessing

Exploratory Data Analysis (EDA)

Feature Selection and Splitting

Building Models (Linear Regression and Deep Neural Network)

Model Evaluation

Conclusion

ASSIGNMENT 1: Boston Housing Price Prediction

1. Problem Statement

The task is to predict the median price of owner-occupied houses in Boston suburbs using 13 different features.

Two approaches are used:

Linear Regression

Deep Neural Network (DNN)

2. About the Dataset

Source: UCI Machine Learning Repository (1978 Boston Housing Data)

Samples: 506 data points (after cleaning, slightly fewer)

Features (13 predictors):

CRIM (Crime rate)

ZN (Residential land proportion)

INDUS (Non-retail business area proportion)

CHAS (Charles River dummy variable: 1 if tract bounds river; 0 otherwise)

NOX (Nitric oxides concentration)

RM (Average rooms per dwelling)

AGE (Proportion of houses built before 1940)

DIS (Distance to employment centers)

RAD (Accessibility to radial highways)

TAX (Property tax rate)

PTRATIO (Pupil-teacher ratio)

B (Proportion of African American residents, calculated)

LSTAT (Lower status population %)

Target Variable:

PRICE (Median value of homes in \$1000s)

3. Data Preprocessing

Loading Data: Using `pandas.read_csv()` with whitespace delimiters.

Initial Checks:

Viewed using `.head()`

Checked dimensions with `np.shape(df)`

Summarized statistics with `.describe()`

No missing values (`df.isnull().sum()` shows zero).

Outlier Detection:

Used boxplots (seaborn) to visually detect outliers.

Quantified outliers in each feature using the Interquartile Range (IQR) method.

Outlier Treatment:

Removed records where $PRICE \geq 35$ because very high prices were considered outliers for this assignment.

4. Exploratory Data Analysis (EDA)

Distribution of Target Variable:

Plotted a histogram for PRICE — found approximately normal distribution (no need for transformations).

Boxplots:

Boxplot visualization confirmed the presence of outliers in several features.

Correlation Analysis:

Created a correlation matrix (`df.corr()`) and plotted using heatmap (seaborn).

Important correlations:

RM (average rooms) positively correlated to PRICE (+0.7 approx)

LSTAT (lower status population %) negatively correlated to PRICE (-0.75 approx)

PTRATIO (pupil-teacher ratio) negatively correlated (-0.5 approx)

Scatterplots:

For the top 3 correlated features (LSTAT, RM, PTRATIO), plotted scatter plots to visualize their relation with the target PRICE.

5. Feature Preparation

Independent Variables (X): All columns except PRICE.

Dependent Variable (y): PRICE

Splitting Dataset:

Train/Test Split: 80% training, 20% testing

`train_test_split()` with `random_state=4` for reproducibility

6. Model 1: Linear Regression

Used `sklearn.linear_model.LinearRegression`

Training:

`.fit(X_train, y_train)`

Prediction:

`.predict(X_test)`

Evaluation Metrics:

Root Mean Squared Error (RMSE):

RMSE

=

1

n

\sum

i

=

1

n

(

y

i

-

y

^

i

)

2

RMSE=

n

1

i=1

\sum

n

(y

i

-

y

^

$\sum_{i=1}^n$

R² Score (Coefficient of Determination):

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

SS_{res}

Where:

SS_{res}

= sum of squared residuals

SS

S
tot
SS
tot

= total sum of squares

7. Model 2: Deep Neural Network (DNN) Why DNN:

To capture nonlinear relationships in the data.

Libraries Used:

keras (specifically Sequential model)

Model Architecture

Layer Details

Input Layer 13 features

Dense Layer 1 128 neurons, activation = ReLU

Dense Layer 2 264 neurons, activation = ReLU

Dense Layer 3 32 neurons, activation = ReLU

Dense Layer 4 16 neurons, activation = ReLU

Output Layer 1 neuron, no activation (for regression)

Activation Functions:

ReLU for hidden layers

No activation (linear) for output layer (regression task)

Optimizer:

adam (adaptive moment estimation, combines momentum and RMSprop)

Loss Function:

mean_squared_error (MSE)

MSE

=

1

n

\sum

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

$$\sum_{i=1}^n$$

$$(y_i - \hat{y}_i)^2$$

$$\sum_{i=1}^n$$

Training:

`.fit(X_train, y_train, epochs=100)`

Data Scaling:

Used StandardScaler to normalize inputs (zero mean and unit variance) because DNNs are sensitive to input scales.

8. Model Evaluation

Predictions:

```
.predict(X_test)
```

Metrics:

R^2 Score (for DNN)

RMSE (for DNN)

Comparison:

DNN achieved a lower RMSE and higher R^2 compared to Linear Regression, indicating a better fit to the data.

9. Conclusion

Linear Regression provided a good baseline.

Deep Neural Network significantly improved the model's performance.

Encouraged to perform:

Hyperparameter Tuning (e.g., neurons per layer, number of layers, learning rate)

Regularization (Dropout layers, L2 penalties)

Early stopping to avoid overfitting.

Important Observations:

Step	Linear Regression	Deep Neural Network
Assumptions	Assumes linearity between features and target	Can learn nonlinear patterns
Data Scaling	Not mandatory	Mandatory (StandardScaler)
Feature Engineering	Limited	Deep learning can find latent interactions
Performance	Good	Better
Complexity	Low	Higher

Final Notes

Training the neural network longer or tuning learning rates could further enhance performance.

Feature selection or dimensionality reduction (PCA) can be explored in future extensions.

Adding dropout layers can make the model more robust to overfitting.

Would you also like me to create a flow diagram or a visual step-by-step workflow for this assignment to make it even clearer? 🚀
(It's especially useful if you have to present this.)

You said:

```
###  
# Install libraries if not already installed  
!pip install tensorflow matplotlib numpy  
###  
# Import necessary libraries  
import tensorflow as tf  
import numpy as np  
import matplotlib.pyplot as plt  
  
###  
# Load Fashion MNIST dataset  
fashion_mnist = tf.keras.datasets.fashion_mnist  
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()  
###  
# Class names for the dataset  
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',  
'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']  
###  
train_images.shape  
###  
len(train_labels)  
  
###  
train_labels  
###  
test_images.shape  
  
###  
len(test_labels)  
  
### md  
# preprocess data  
###  
plt.figure()  
plt.imshow(train_images[0])  
plt.colorbar()  
plt.grid(False)  
plt.show()  
###
```

```

# Normalize the images to [0,1]
train_images = train_images / 255.0
test_images = test_images / 255.0
###
# Display the first 25 images from the training dataset along with their labels
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
###
# Build the neural network model
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)), # Flatten the image (28x28 to 784)
    tf.keras.layers.Dense(128, activation='relu'), # Fully connected layer with 128 neurons
    tf.keras.layers.Dense(10) # Output layer with 10 neurons (one for each class)
])
###
# Compile the model
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

###
# Train the model and capture the training history
history = model.fit(train_images, train_labels, epochs=30)
###
# Evaluate the model's performance on the test set
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print(f'\nTest accuracy: {test_acc}')
###
# Make predictions on the test data
probability_model = tf.keras.Sequential([model, tf.keras.layers.Softmax()])
predictions = probability_model.predict(test_images)
###
# Function to plot the image and its prediction
def plot_image(i, predictions_array, true_label, img):
    true_label, img = true_label[i], img[i]
    plt.grid(False)
    plt.xticks([])

```

```

plt.yticks([])
plt.imshow(img, cmap=plt.cm.binary)
predicted_label = np.argmax(predictions_array)
color = 'blue' if predicted_label == true_label else 'red'
plt.xlabel(f"{class_names[predicted_label]} {100*np.max(predictions_array):2.0f}%
({class_names[true_label]})", color=color)
###
# Function to plot the prediction values
def plot_value_array(i, predictions_array, true_label):
    true_label = true_label[i]
    plt.grid(False)
    plt.xticks(range(10))
    plt.yticks([])
    thisplot = plt.bar(range(10), predictions_array, color="#777777")
    plt.ylim([0, 1])
    predicted_label = np.argmax(predictions_array)
    thisplot[predicted_label].set_color('red')
    thisplot[true_label].set_color('blue')
###
# Visualize the predictions on the test data
i = 0
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i], test_labels)
plt.show()
###
# Let's look at more images and their predictions
i = 12
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i], test_labels)
plt.show()

###
# Using the trained model to make predictions on a single image
img = test_images[1]
print(f"Image shape: {img.shape}")
###
# Expand dimensions to match the model's input shape (batch of 1)
img = np.expand_dims(img, 0)

```



```

predictions_single = probability_model.predict(img)
#%%
# Plot the prediction and its confidence values
plot_value_array(1, predictions_single[0], test_labels)
_ = plt.xticks(range(10), class_names, rotation=45)
plt.show()
#%%
# Output the predicted label
print(f"Predicted label: {np.argmax(predictions_single[0])}")
#%%
# Plot a grid of the first X test images with their predictions
num_rows = 5
num_cols = 3
num_images = num_rows * num_cols
plt.figure(figsize=(2 * 2 * num_cols, 2 * num_rows))
for i in range(num_images):
    plt.subplot(num_rows, 2 * num_cols, 2 * i + 1)
    plot_image(i, predictions[i], test_labels, test_images)
    plt.subplot(num_rows, 2 * num_cols, 2 * i + 2)
    plot_value_array(i, predictions[i], test_labels)
plt.tight_layout()
plt.show()

```

ChatGPT said:

Your code is very well-structured and covers a full workflow for training and evaluating a simple neural network on the Fashion MNIST dataset using TensorFlow and Matplotlib!

Here's a summary of what your code achieves:

Install necessary libraries if missing.

Load and preprocess the Fashion MNIST dataset.

Visualize the raw and normalized data.

Build a Sequential Neural Network:

Flatten the 28×28 images.

Add a dense hidden layer with 128 neurons and ReLU activation.

Add a dense output layer with 10 neurons (no activation yet).

Compile the model using:

Adam optimizer

SparseCategoricalCrossentropy loss (with logits)

Train the model for 30 epochs.

Evaluate the model on the test set.

Make predictions and visualize:

Show the test image alongside the model's prediction and the confidence scores.

Display multiple images and predictions in a grid.

A few minor improvements (optional):

Loss function / activation at the end:

You are using `from_logits=True` in the loss, which is correct because your model output is raw logits (no activation function on last layer).

Later you create `probability_model` by adding Softmax to get proper probabilities for predictions.

✅ This is correct, but alternatively you could have also directly added `activation='softmax'` to your final dense layer and used `from_logits=False` while compiling. (Both approaches are valid.)

Plotting a single prediction:

In this block:

```
plot_value_array(1, predictions_single[0], test_labels)
```

You're plotting prediction for test image 1, but you should actually plot for index 0 because `img = np.expand_dims(img, 0)` creates a batch of 1 image (index 0).

So change to:

```
plot_value_array(0, predictions_single[0], test_labels)
```

Training for 30 epochs:

30 epochs may be a little too much for this simple model — you might notice overfitting after about 10-15 epochs (training accuracy keeps increasing but test accuracy plateaus).

You could plot the training history:

```
plt.plot(history.history['accuracy'])
```

```
plt.title('Model accuracy')
```

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

```
plt.xlabel('Epoch')
```

```
plt.show()
```

— to visually check when it starts overfitting.

Overall:

Code Quality: Good modular blocks and documentation.

Understanding: Shows a clear grasp of model building, training, evaluation, and visualization in TensorFlow.

Would you like me to also show you how to improve this model's performance slightly by:

Adding Dropout to reduce overfitting?

Using a deeper or more advanced model?

Or plotting training vs validation loss/accuracy curves?

Let me know! 🚀

You said:

explain each thing, and also the functions, the diagrams and how to read them

ChatGPT said:

Got it — you want a full explanation:

Each part of the code,

Functions you wrote,

Diagrams (like image plots, prediction bar charts),

How to interpret what you see.

I'll explain it in a detailed, step-by-step way suitable for building real understanding.

🧠 Part 1: Installing and Importing

`!pip install tensorflow matplotlib numpy`

TensorFlow: Library for building and training machine learning models.

Matplotlib: Library for plotting graphs and images.

NumPy: Library for numerical operations, mainly arrays.

```
import tensorflow as tf
```

```
import numpy as np
```

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

Here you're importing these libraries into your Python code.

🧠 Part 2: Loading Fashion MNIST

```
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
Fashion MNIST is a dataset of 28x28 grayscale images of clothes (like shirts, shoes, bags).
```

You split it into:

Training set (train_images, train_labels) — used to train the model.

Test set (test_images, test_labels) — used to evaluate performance after training.

🧠 Part 3: Understanding the Data

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

train_labels are numbers (0-9), where each number corresponds to a clothing item.

This class_names list helps map numbers to human-readable labels.

🧠 Part 4: Data Preprocessing

Plot a sample image:

```
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
imshow(train_images[0]): Shows the first image.
```

colorbar(): Adds a color scale next to the image (to see pixel values).

grid(False): Removes grid lines for clean view.

Normalize the pixel values:

```
train_images = train_images / 255.0
test_images = test_images / 255.0
Original pixel values are 0–255.
```

Normalize to 0–1 range so the model trains faster and more reliably.

🧠 Part 5: Visualizing the dataset

Plot 25 training images:

```
plt.figure(figsize=(10,10))
for i in range(25):
```

```
plt.subplot(5, 5, i + 1)
plt.xticks([])
plt.yticks([])
plt.grid(False)
plt.imshow(train_images[i], cmap=plt.cm.binary)
plt.xlabel(class_names[train_labels[i]])
plt.show()
subplot(5, 5, i+1): Create a 5×5 grid of images.
```

imshow(): Show each image.

xlabel(): Label with the clothing class.



How to read this diagram:

Each small image is a training example.

Below each image is its true label (like 'Shirt', 'Sandal').



Part 6: Building the Model

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])
```

Layer-by-layer explanation:

Flatten: Converts the 2D image (28x28) into 1D array (784 numbers).

Dense(128, activation='relu'): A hidden layer with 128 neurons.

ReLU activation: Helps model learn complex patterns.

Dense(10): Final output layer with 10 neurons (one for each clothing class).



Part 7: Compiling the Model

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Optimizer: 'adam' adjusts weights to minimize loss.

Loss function: Measures how far predictions are from true labels.

Accuracy: Track the fraction of correct predictions.

🧠 Part 8: Training the Model

```
history = model.fit(train_images, train_labels, epochs=30)
```

Train the model for 30 rounds (epochs) on the training data.

history will store the loss and accuracy after each epoch.

🧠 Part 9: Evaluating the Model

```
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
```

Evaluate how well your model performs on unseen data (test set).

Accuracy you print here is a measure of how good your model is.

🧠 Part 10: Making Predictions

```
probability_model = tf.keras.Sequential([model, tf.keras.layers.Softmax()])
```

```
predictions = probability_model.predict(test_images)
```

Softmax layer converts model outputs into probabilities.

predictions[i] gives probability for each of the 10 classes for image i.

🧠 Part 11: Plotting Results

a. plot_image function

```
def plot_image(i, predictions_array, true_label, img):
```

```
    ...
```

Inputs:

i: Index of the image.

predictions_array: Predicted probabilities.

true_label: True label from dataset.

img: Image array.

What it does:

Displays the image.

Shows predicted label, confidence %, and true label.

Blue color if prediction is correct, red if wrong.

✅ How to read the plot:

Title shows: Predicted class (confidence%) (True class).

Blue = correct, Red = incorrect.

b. plot_value_array function

```
def plot_value_array(i, predictions_array, true_label):
```

```
    ...
```

Inputs:

Same as above.

What it does:

Displays a bar graph of prediction probabilities for all 10 classes.

The bar for:

Predicted label is RED.

True label is BLUE.

✅ How to read the plot:

Each bar = probability for a class.

The highest red bar = model's guess.

True class should ideally have the highest bar.

🧠 Part 12: Predicting a Single Image

```
img = np.expand_dims(img, 0)
```

```
predictions_single = probability_model.predict(img)
```

Expand the dimensions so it matches batch size (even for 1 image).

Then predict.

✅ How to read the single image prediction:

Just like before — look at bar heights.

The label with the highest confidence is the predicted class.

🧠 Part 13: Grid of Predictions

```
num_rows = 5
```

```
num_cols = 3
```

...

for i in range(num_images):

...

Display multiple predictions (images and bar charts) together.

✅ How to read:

Each image is immediately followed by its prediction bar chart.

Quickly spot where model is right or wrong.

🧠 Overall Flow Diagram (Simplified)

Fashion MNIST Data → Preprocess → Build Model → Train Model → Evaluate on Test Set → Predict → Visualize Results

✅ Summary

Step What you did How to understand it

Load Data Got images and labels Each image = clothing

Preprocess Normalize images Scale pixel values 0-1

Build Model Flatten + Dense + Dense Neural Network

Compile Adam + Loss Prepare for training

Train Fit for 30 epochs Model learns from data

Evaluate Test set performance Real-world test

Predict Probabilities for classes Softmax output

Visualize Images + Bar graphs Easy to spot correct/wrong predictions

Would you also like me to draw a simple concept diagram of the Neural Network architecture you built (Flatten → Dense → Dense) to make it even clearer? 🚀

(If yes, I'll create a clean visual for you!)

You said:

```
###
```

```
# Install libraries if not already installed
```

```
!pip install tensorflow matplotlib numpy
```

```
###
```

```
# Import necessary libraries
```

```
import tensorflow as tf
```

```
import numpy as np
```

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

```
###
```

```
# Load Fashion MNIST dataset
```

```
fashion_mnist = tf.keras.datasets.fashion_mnist
```

```
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

```
###
```



```

# Class names for the dataset
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
#%%
train_images.shape
#%%
len(train_labels)

#%%
train_labels
#%%
test_images.shape

#%%
len(test_labels)

#%% md
# preprocess data
#%%
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
#%%
# Normalize the images to [0,1]
train_images = train_images / 255.0
test_images = test_images / 255.0
#%%
# Display the first 25 images from the training dataset along with their labels
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
#%%
# Build the neural network model
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)), # Flatten the image (28x28 to 784)
    tf.keras.layers.Dense(128, activation='relu'), # Fully connected layer with 128 neurons

```

```

    tf.keras.layers.Dense(10) # Output layer with 10 neurons (one for each class)
])
###
# Compile the model
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

###
# Train the model and capture the training history
history = model.fit(train_images, train_labels, epochs=30)
###
# Evaluate the model's performance on the test set
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print(f'\nTest accuracy: {test_acc}')
###
# Make predictions on the test data
probability_model = tf.keras.Sequential([model, tf.keras.layers.Softmax()])
predictions = probability_model.predict(test_images)
###
# Function to plot the image and its prediction
def plot_image(i, predictions_array, true_label, img):
    true_label, img = true_label[i], img[i]
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(img, cmap=plt.cm.binary)
    predicted_label = np.argmax(predictions_array)
    color = 'blue' if predicted_label == true_label else 'red'
    plt.xlabel("{}{class_names[predicted_label]} {100*np.max(predictions_array):2.0f}%
({class_names[true_label]})", color=color)
###
# Function to plot the prediction values
def plot_value_array(i, predictions_array, true_label):
    true_label = true_label[i]
    plt.grid(False)
    plt.xticks(range(10))
    plt.yticks([])
    thisplot = plt.bar(range(10), predictions_array, color="#777777")
    plt.ylim([0, 1])
    predicted_label = np.argmax(predictions_array)
    thisplot[predicted_label].set_color('red')
    thisplot[true_label].set_color('blue')
###

```

```

# Visualize the predictions on the test data
i = 0
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i], test_labels)
plt.show()
###
# Let's look at more images and their predictions
i = 12
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i], test_labels)
plt.show()

###
# Using the trained model to make predictions on a single image
img = test_images[1]
print(f"Image shape: {img.shape}")
###
# Expand dimensions to match the model's input shape (batch of 1)
img = np.expand_dims(img, 0)
predictions_single = probability_model.predict(img)
###
# Plot the prediction and its confidence values
plot_value_array(1, predictions_single[0], test_labels)
_ = plt.xticks(range(10), class_names, rotation=45)
plt.show()
###
# Output the predicted label
print(f"Predicted label: {np.argmax(predictions_single[0])}")
###
# Plot a grid of the first X test images with their predictions
num_rows = 5
num_cols = 3
num_images = num_rows * num_cols
plt.figure(figsize=(2 * 2 * num_cols, 2 * num_rows))
for i in range(num_images):
    plt.subplot(num_rows, 2 * num_cols, 2 * i + 1)
    plot_image(i, predictions[i], test_labels, test_images)
    plt.subplot(num_rows, 2 * num_cols, 2 * i + 2)

```

```
    plot_value_array(i, predictions[i], test_labels)
plt.tight_layout()
plt.show()
#%%
```

ChatGPT said:

This script runs through the process of loading, preprocessing, training, and evaluating a Neural Network model on the Fashion MNIST dataset. Let's break it down step-by-step:

1. Installing Libraries

```
!pip install tensorflow matplotlib numpy
```

TensorFlow: Deep learning framework used to build and train the neural network.

Matplotlib: Used for plotting visualizations (images, charts).

NumPy: Performs numerical operations, especially working with arrays.

2. Import Libraries

```
import tensorflow as tf
```

```
import numpy as np
```

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

These are the libraries you use for creating and training models, and visualizing data.

3. Load Fashion MNIST Dataset

```
fashion_mnist = tf.keras.datasets.fashion_mnist
```

```
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

Fashion MNIST: A dataset containing 28x28 grayscale images of 10 clothing classes.

Split into training and test sets.

4. Class Names

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',  
'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

Class names represent the categories the images belong to. Each number (0-9) maps to one of these labels.

5. Data Exploration

```
train_images.shape
```

```
len(train_labels)
```

```
train_labels
```

```
test_images.shape
```

```
len(test_labels)
```

`train_images.shape`: Returns the shape of the training image dataset.

Should output (60000, 28, 28) (60,000 images of size 28x28 pixels).

`len(train_labels)`: Gives the number of labels in the training set (60,000).

`train_labels`: The actual labels for the training set.

`test_images.shape`: The test set (10,000 images of 28x28).

`len(test_labels)`: The number of labels in the test set.

6. Visualizing the First Image

```
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```

This displays the first image from the training set.

The colorbar gives a scale for pixel intensity values.

No grid is shown on the image for better clarity.

7. Normalize the Image Data

```
train_images = train_images / 255.0
test_images = test_images / 255.0
```

Pixel values range from 0 to 255.

Normalize to the range 0 to 1 for easier processing by the model.

8. Visualize Multiple Images

```
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

Grid of 25 images from the training set.

Each image is labeled with its true class (from `class_names`).

9. Build the Neural Network Model

```
model = tf.keras.Sequential([
```

```
tf.keras.layers.Flatten(input_shape=(28, 28)), # Flatten the image (28x28 to 784)
tf.keras.layers.Dense(128, activation='relu'), # Fully connected layer with 128 neurons
tf.keras.layers.Dense(10) # Output layer with 10 neurons (one for each class)
])
```

Flatten: Converts 2D images into a 1D array (784 values per image).

Dense(128): A hidden layer with 128 neurons and ReLU activation.

Dense(10): The output layer with 10 neurons (for each class).

10. Compile the Model

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

optimizer='adam': An optimization algorithm that adjusts weights.

loss: SparseCategoricalCrossentropy is used for multi-class classification (labels are integers).

metrics: We track accuracy as the model's performance metric.

11. Train the Model

```
history = model.fit(train_images, train_labels, epochs=30)
```

Training the model for 30 epochs on the training data (train_images and train_labels).

history stores the training history (loss, accuracy, etc.).

12. Evaluate the Model

```
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
```

```
print(f'\nTest accuracy: {test_acc}')
```

Evaluates how well the trained model performs on the test set.

Prints the test accuracy.

13. Predictions

```
probability_model = tf.keras.Sequential([model, tf.keras.layers.Softmax()])
```

```
predictions = probability_model.predict(test_images)
```

Softmax layer is added to convert raw outputs into probabilities.

predictions stores the probability for each class for each test image.

14. Plotting Functions

Plot Image with Prediction

```
def plot_image(i, predictions_array, true_label, img):
```

```
...
```

Visualizes an image along with:

Predicted class and its confidence (in %).

True label.

Blue if correct, red if incorrect.

Plot Prediction Values (Bar chart)

```
def plot_value_array(i, predictions_array, true_label):
```

```
    ...
```

Visualizes the predicted probability for each class (0-9) as a bar chart.

Highlights the predicted class in red and true class in blue.

15. Visualizing Predictions

```
i = 0
```

```
plt.figure(figsize=(6,3))
```

```
plt.subplot(1,2,1)
```

```
plot_image(i, predictions[i], test_labels, test_images)
```

```
plt.subplot(1,2,2)
```

```
plot_value_array(i, predictions[i], test_labels)
```

```
plt.show()
```

Displays an image from the test set and its prediction (as a bar chart).

16. More Predictions

```
i = 12
```

```
plt.figure(figsize=(6,3))
```

```
plt.subplot(1,2,1)
```

```
plot_image(i, predictions[i], test_labels, test_images)
```

```
plt.subplot(1,2,2)
```

```
plot_value_array(i, predictions[i], test_labels)
```

```
plt.show()
```

Visualizes another test image with its predicted probabilities.

17. Making Predictions on a Single Image

```
img = test_images[1]
```

```
print(f"Image shape: {img.shape}")
```

```
img = np.expand_dims(img, 0)
```

```
predictions_single = probability_model.predict(img)
```

Selects one image from the test set.

Expands it to match the model's input shape (batch of size 1).

Predicts the label for this image.

18. Plot Prediction for a Single Image

```
plot_value_array(1, predictions_single[0], test_labels)
_ = plt.xticks(range(10), class_names, rotation=45)
plt.show()
```

Displays the prediction bar chart for this image.

19. Output Predicted Label

```
print(f'Predicted label: {np.argmax(predictions_single[0])}')

```

Prints the predicted class (the class with the highest probability).

20. Plot Multiple Predictions

```
num_rows = 5
num_cols = 3
num_images = num_rows * num_cols
plt.figure(figsize=(2 * 2 * num_cols, 2 * num_rows))
for i in range(num_images):
```

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

Displays a grid of multiple predictions from the test set with images and their prediction bar charts.

Interpretation:

Blue bar in prediction charts means the model was correct.

Red bar means the model was wrong.

By examining these plots, you can analyze the performance of the model visually.

Would you like a summary table with key outputs of each step, or a concept diagram of the model architecture for better understanding?

You said:

```
#%% md
```

```
Import Packages
```

```
#%%
```

```
import numpy as np
```

```
from keras.datasets import imdb
```

```
from keras import models
```

```
from keras import layers
```

```
from keras import optimizers
```

```
from keras import losses
```



```
from keras import metrics
```

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

```
%matplotlib inline
```

```
#### md
```

```
Loading the Data
```

```
####
```

```
# Load the data, keeping only 10,000 of the most frequently occurring words
```

```
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words = 10000)
```

```
####
```

```
train_data[:2]
```

```
####
```

```
train_labels
```

```
####
```

```
# Check the first label
```

```
train_labels[0]
```

```
####
```

```
# Since we restricted ourselves to the top 10000 frequent words, no word index should exceed 10000
```

```
# we'll verify this below
```

```
# Here is a list of maximum indexes in every review --- we search the maximum index in this list of max indexes
```

```
print(type([max(sequence) for sequence in train_data]))
```

```
# Find the maximum of all max indexes
```

```
max([max(sequence) for sequence in train_data])
```

```
####
```

```
# Let's quickly decode a review
```

```
# step 1: load the dictionary mappings from word to integer index
```

```
word_index = imdb.get_word_index()
```

```
# step 2: reverse word index to map integer indexes to their respective words
```

```
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
```

```
# Step 3: decode the review, mapping integer indices to words
```

```
#
```

```
# indices are off by 3 because 0, 1, and 2 are reserved indices for "padding", "Start of sequence" and "unknown"
```

```
decoded_review = ' '.join([reverse_word_index.get(i-3, '?') for i in train_data[0]])
```

```
decoded_review
```

```

#%%
len(reverse_word_index)
#%% md
Preparing the data
#%%
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension)) # Creates an all zero matrix of shape
    (len(sequences),10K)
    for i,sequence in enumerate(sequences):
        results[i,sequence] = 1 # Sets specific indices of results[i] to 1s
    return results

# Vectorize training Data
X_train = vectorize_sequences(train_data)

# Vectorize testing Data
X_test = vectorize_sequences(test_data)
#%%
X_train[0]
#%%
X_train.shape
#%% md
Vectorize labels
#%%
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
#%% md
Model defination
#%%
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
#%%
model.compile(
    optimizer=optimizers.RMSprop(learning_rate=0.001),
    loss = losses.binary_crossentropy,
    metrics = [metrics.binary_accuracy]
)
#%%
# Input for Validation
X_val = X_train[:10000]
partial_X_train = X_train[10000:]

```

```

# Labels for validation
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
### md
Training our model
###
history = model.fit(
    partial_X_train,
    partial_y_train,
    epochs=20,
    batch_size=512,
    validation_data=(X_val, y_val)
)
###
history_dict = history.history
history_dict.keys()
###
# Plotting losses
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']

epochs = range(1, len(loss_values) + 1)

plt.plot(epochs, loss_values, 'g', label="Training Loss")
plt.plot(epochs, val_loss_values, 'b', label="Validation Loss")

plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss Value')
plt.legend()

plt.show()
###
# Training and Validation Accuracy

acc_values = history_dict['binary_accuracy']
val_acc_values = history_dict['val_binary_accuracy']

epochs = range(1, len(loss_values) + 1)

plt.plot(epochs, acc_values, 'g', label="Training Accuracy")
plt.plot(epochs, val_acc_values, 'b', label="Validation Accuracy")

plt.title('Training and Validation Accuraccy')

```

```

plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
### md
Retraining our model
###
model.fit(
    partial_X_train,
    partial_y_train,
    epochs=3,
    batch_size=512,
    validation_data=(X_val, y_val)
)
### md
Model Evaluation
###
# Making Predictions for testing data
np.set_printoptions(suppress=True)
result = model.predict(X_test)
###
result
###
y_pred = np.zeros(len(result))
for i, score in enumerate(result):
    y_pred[i] = np.round(score)
###
mae = metrics.mean_absolute_error(y_pred, y_test)
mae
###

```

ChatGPT said:

The code you provided trains and evaluates a neural network model for sentiment analysis using the IMDB dataset. Here's a breakdown of the key sections:

Data Loading:

The dataset is loaded using Keras' `imdb.load_data()` method, with the top 10,000 most frequent words considered.

The data consists of sequences of integers, where each integer represents a word.

Decoding Reviews:

The reviews are decoded using a word index that maps integers to words.

Preparing Data:

`vectorize_sequences()` function is used to convert the sequences of integers into a binary matrix, where each row corresponds to a review, and the columns represent the words.

Model Definition:

A Sequential model is created with two hidden layers of 16 neurons each and a sigmoid output layer for binary classification.

Compiling the Model:

The model is compiled with RMSprop optimizer, binary cross-entropy loss, and binary accuracy metrics.

Training the Model:

The model is trained on a subset of the training data (`partial_X_train` and `partial_y_train`), and validation data (`X_val` and `y_val`) is used to monitor performance during training.

Visualizing Training History:

The loss and accuracy during training and validation are plotted to visualize the model's performance.

Retraining:

The model is retrained for an additional 3 epochs to further improve performance.

Model Evaluation:

The model is used to predict on the test data (`X_test`), and the predictions are compared with the actual labels to compute metrics like Mean Absolute Error (MAE).

Let me know if you need any specific explanations or modifications!