HPC Practicals

Assignment 1

Q. Design and implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS and DFS.

To explain and implement **parallel Breadth-First Search (BFS)** and **Depth-First Search (DFS)** using **OpenMP**, let’s break it down into the **theory**, followed by **code implementation** using a **graph (adjacency list)** structure. We'll go with an **undirected graph** to generalize the application.

Theory of Design

Graph Representation

We'll use an **adjacency list** to represent an undirected graph. Each node will keep a list of its neighbors.

Breadth-First Search (BFS)

Sequential BFS

- Uses a **queue** to explore nodes level by level.
- Start at a source node, explore its neighbors, then the neighbors' neighbors, etc.

Parallel BFS (with OpenMP)

- At each **level**, all nodes in the current frontier can be explored **in parallel**.
- Use OpenMP to parallelize the loop that iterates through current-level nodes.

BFS is more naturally parallelizable because each level can be processed in parallel.

Depth-First Search (DFS)

Sequential DFS

- Uses a **stack** (or recursion) to go deep along one branch before backtracking.

Parallel DFS (with OpenMP)

- DFS is inherently recursive and not easy to parallelize because it explores deep paths sequentially.
- We can spawn threads from the root for each unvisited neighbor (i.e., parallelize top-level recursion).

Parallel DFS usually leverages **task-based parallelism**.

Code

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

class Graph {
public:
    int V;
    vector<vector<int>> adj;

    Graph(int V) {
        this->V = V;
        adj.resize(V);
    }

    void addEdge(int u, int v) {
        adj[u].push_back(v);
        adj[v].push_back(u);  // Undirected graph
    }

    void parallelBFS(int start) {
        vector<bool> visited(V, false);
        queue<int> q;

        visited[start] = true;
        q.push(start);

        while (!q.empty()) {
            int qSize = q.size();
            vector<int> frontier;

            // Collect current level nodes
            for (int i = 0; i < qSize; ++i) {
                int node = q.front(); q.pop();
                cout << node << " ";

                frontier.push_back(node);
            }

            // Explore neighbors in parallel
            #pragma omp parallel for schedule(dynamic)
            for (int i = 0; i < frontier.size(); ++i) {
                int u = frontier[i];
                for (int v : adj[u]) {
                    if (!visited[v]) {
                        #pragma omp critical
                        {
                            if (!visited[v]) {
                                visited[v] = true;
                                q.push(v);
                            }
                        }
                    }
                }
            }
        }
        cout << endl;
    }
```

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

        #pragma omp parallel for schedule(dynamic)
        for (int i = 0; i < adj[node].size(); ++i) {
            int neighbor = adj[node][i];
            if (!visited[neighbor]) {
                #pragma omp task firstprivate(neighbor)
                {
                    parallelDFSUtil(neighbor, visited);
                }
            }
        }
        #pragma omp taskwait
    }

    void parallelDFS(int start) {
        vector<bool> visited(V, false);
        #pragma omp parallel
        {
            #pragma omp single
            {
                parallelDFSUtil(start, visited);
            }
        }
        cout << endl;
    }
};

int main() {
    Graph g(7);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 5);
    g.addEdge(2, 6);

    cout << "Parallel BFS starting from node 0:\n";
    g.parallelBFS(0);

    cout << "Parallel DFS starting from node 0:\n";
    g.parallelDFS(0);

    return 0;
}
```

Header Files

```
#include <iostream>     // For input/output
#include <vector>       // To use dynamic arrays (adjacency list)
#include <queue>        // For BFS (FIFO structure)
#include <stack>        // (Not used here but typical for DFS)
#include <omp.h>        // OpenMP for parallel programming
```

Graph Class Definition

```
class Graph {
public:
    int V;                          // Number of vertices
    vector<vector<int>> adj;        // Adjacency list for graph
```

Constructor and Edge Addition

```
    Graph(int V) {
        this->V = V;
        adj.resize(V);              // Create a list for each vertex
    }

    void addEdge(int u, int v) {
        adj[u].push_back(v);
        adj[v].push_back(u);        // Because it's an undirected graph
    }
```

Parallel Breadth-First Search (BFS)

```
 void parallelBFS(int start) {
        vector<bool> visited(V, false);  // Keep track of visited nodes
        queue<int> q;

        visited[start] = true;      // Mark the start node as visited
        q.push(start);              // Start BFS from this node
```

While queue isn't empty

```
        while (!q.empty()) {
            int qSize = q.size();           // Number of nodes at current level
            vector<int> frontier;
            // Step 1: Extract current level
            for (int i = 0; i < qSize; ++i) {
                int node = q.front(); q.pop();
                cout << node << " ";        // Process the node
                frontier.push_back(node);  // Save for parallel exploration
            }
```

Step 2: Explore neighbors in parallel

```
            #pragma omp parallel for schedule(dynamic)
            for (int i = 0; i < frontier.size(); ++i) {
                int u = frontier[i];
                for (int v : adj[u]) {
                    if (!visited[v]) {
                        #pragma omp critical
                        {
                            if (!visited[v]) {
                                visited[v] = true;
                                q.push(v);
                            }
                        }
                    }
                }
            }
        }
        cout << endl;
    }
```

What's going on:

- **Frontier**: Nodes in the current BFS level.
- **Parallel for**: Multiple threads explore neighbors of current-level nodes.
- **Critical section**: Protects concurrent writes to `visited` and `queue` to avoid race conditions.

`#pragma omp parallel for schedule(dynamic)`

1. `#pragma omp`

This tells the compiler:

> "Hey, OpenMP — I want to do something in parallel here."

It's a **compiler directive**.

2. `parallel for`

This tells OpenMP:

> "Run the following `for` loop in parallel — divide the iterations among available threads."

Each thread will handle a portion of the loop iterations.

3. `schedule(dynamic)`

This controls **how** the loop iterations are divided up and assigned to threads.

Dynamic Scheduling:

- Iterations are **not pre-assigned** at the start.
- Instead, OpenMP maintains a **work queue**.
- Each thread **requests a chunk** of iterations when it finishes its current chunk.
- This is useful when iterations take **uneven or unpredictable amounts of time** (like quicksort, I/O, simulations, etc.).

What it looks like:

If you have:

```
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < 1000; i++) {
    do_work(i);
}
```

Then OpenMP might assign:

- Thread 0 → iterations 0–9
- Thread 1 → 10–19
- Once a thread finishes its chunk, it grabs another from the pool.

This continues until all iterations are done.

Why use `schedule(dynamic)`?

- **Load balancing**: Useful when iterations vary in workload. Threads that finish early don't sit idle — they grab more work.
- Prevents **thread starvation** (some threads doing all the work while others finish early).

Optional Chunk Size

You can also specify chunk size like this:

```
#pragma omp parallel for schedule(dynamic, 5)
```

This means:

- Assign chunks of 5 iterations to threads at a time.
- Threads will keep grabbing chunks of 5 from the queue until done.

Recursive DFS with OpenMP Tasks

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

Parallel for-loop over neighbors

```
    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < adj[node].size(); ++i) {
        int neighbor = adj[node][i];
        if (!visited[neighbor]) {
            #pragma omp task firstprivate(neighbor)
            {
                parallelDFSUtil(neighbor, visited);
            }
        }
    }
    #pragma omp taskwait   // Wait for all tasks to finish
}
```

What's happening:

- DFS is **recursive**, and we use `#pragma omp task` to run each neighbor's DFS in parallel.
- `firstprivate(neighbor)`: Ensures each task gets its own copy of `neighbor`.
- `taskwait`: Makes sure all recursive branches complete before returning.

DFS Entry Point

```
void parallelDFS(int start) {
    vector<bool> visited(V, false);
    #pragma omp parallel
    {
        #pragma omp single
        {
```

```
                parallelDFSUtil(start, visited);
            }
        }
        cout << endl;
    }
```

What this does:

- Starts a **single root task** to kick off DFS.
- The actual parallelism happens in the recursive utility function using **tasks**.

Main Function: Setup and Run

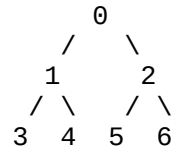
```
int main() {
    Graph g(7);              // Create a graph with 7 nodes

    // Add edges (undirected)
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 5);
    g.addEdge(2, 6);

    cout << "Parallel BFS starting from node 0:\n";
    g.parallelBFS(0);

    cout << "Parallel DFS starting from node 0:\n";
    g.parallelDFS(0);

    return 0;
}
```

Summary of Key OpenMP Features Used

Directive	Purpose
`#pragma omp parallel for`	Parallelizes loops over independent data (e.g., frontier nodes)
`#pragma omp critical`	Protects critical section from concurrent access
`#pragma omp task`	Spawns a new task (useful for recursive DFS)
`#pragma omp single`	Ensures a single thread runs the initial DFS call
`#pragma omp taskwait`	Waits for all tasks launched so far to complete

Graph Structure

This code defines a **simple undirected graph** with 7 nodes (0 to 6):



This is like a binary tree rooted at node `0`.

PART 1: Parallel BFS (Breadth-First Search)

Step-by-step Flow:

1. **Start at node `0`**:
    - Mark it visited.
    - Push it into the queue: `q = [0]`.
2. **Main BFS Loop:**

At each level:

- Extract all nodes from the queue into a vector called `frontier`.
- Traverse all nodes in `frontier` **in parallel**.
- For each neighbor, if it's not visited, mark as visited and add to queue.

Parallelism:

- This part is parallelized:

  ```
  #pragma omp parallel for schedule(dynamic)
  for (int i = 0; i < frontier.size(); ++i) {
  ```

  → Each node in the current **level** (frontier) is processed by a separate thread.
- The `#pragma omp critical` block ensures only **one thread updates shared data** (`visited[]`, `q.push`) at a time — preventing race conditions.

Visualization of Execution

```
Level 0 (Queue: [0]):
→ Thread 1: visits 0 → frontier = [0]
→ Adjacent: 1, 2 → marked visited → Queue = [1, 2]

Level 1 (Queue: [1, 2]):
→ Thread 1: visits 1
→ Thread 2: visits 2
→ Adjacent of 1: 3, 4 → mark visited
→ Adjacent of 2: 5, 6 → mark visited → Queue = [3, 4, 5, 6]

Level 2 (Queue: [3, 4, 5, 6]):
→ Threads in parallel visit 3, 4, 5, 6
→ No new nodes → Done
```

BFS finishes level by level using **dynamic scheduling** (threads grab work as they finish).

PART 2: Parallel DFS (Depth-First Search)

DFS is recursive. We visit a node, then recursively visit its neighbors.

Step-by-step Flow:

1. **Start at node `0`**:
    - Mark it visited.
    - For each neighbor (`1`, `2`), create **OpenMP tasks** to handle them in parallel:

      ```
      cpp
      CopyEdit
      #pragma omp task firstprivate(neighbor)
      {
          parallelDFSUtil(neighbor, visited);
      }
      ```

2. Each recursive call follows the same pattern:
    - Mark current node visited.
    - For each unvisited neighbor → create a new task.

Task-based Parallelism

- Tasks are created dynamically for each neighbor.
- `#pragma omp taskwait` ensures the current task waits for child tasks to finish before continuing.

Visualization of Execution

```
→ Start at 0 (main thread)
  ↳ Create task for 1
      ↳ Create tasks for 3 and 4
  ↳ Create task for 2
      ↳ Create tasks for 5 and 6
```

Each of these branches runs **concurrently**.

Final Output

Output of `parallelBFS(0)`:

(Traversal level-wise, order may slightly vary due to thread execution)

```
0 1 2 3 4 5 6
```

Output of `parallelDFS(0)`:

(Order can vary since DFS is recursive + parallelized)

Example:
```
0 1 3 4 2 5 6
```
Or:
```
0 2 5 6 1 3 4
```
Exact order depends on how OpenMP schedules the tasks

Assignment 2

Write a program to implement Parallel Bubble Sort and Merge sort using OpenMP. Use existing algorithms and measure the performance of sequential and parallel algorithms.

Below is an explanation of the theory behind parallelizing two common sorting algorithms—Bubble Sort (using the odd–even transposition method) and Merge Sort—and sample C++ code that implements both sequential and parallel versions using OpenMP. The code also measures performance to compare the sequential and parallel implementations.

Theory

Bubble Sort and Its Parallel Variant

Sequential Bubble Sort:
Bubble sort repeatedly compares adjacent pairs in an array and swaps them if they are out of order. It performs this pass over the array repeatedly until no swaps are needed. Although simple, it has a worst-case time complexity of $O(n^2)$.

Parallel Bubble Sort (Odd–Even Transposition Sort):
The odd–even transposition sort is a variant of bubble sort that is more amenable to parallelization. It divides the sorting pass into two phases:

- **Odd Phase:** Compare and swap elements at positions (1, 2), (3, 4), …
- **Even Phase:** Compare and swap elements at positions (0, 1), (2, 3), …

Within each phase, comparisons are independent and can be performed concurrently. OpenMP's parallel loops can be used to execute the comparisons in parallel, while a shared flag (with a reduction) helps determine if another pass is needed.

Merge Sort and Its Parallel Variant

Sequential Merge Sort:
Merge sort is a classic divide-and-conquer algorithm. It recursively divides the array into halves until each subarray has one element, then merges the sorted subarrays back together. Its time complexity is $O(n \log n)$.

Parallel Merge Sort:
Merge sort is naturally suited for parallelization since its recursive division of the array can be performed concurrently. By using OpenMP tasks, the two halves of the array can be sorted in parallel. To prevent too many tasks from being spawned (which might lead to overhead), a depth limit is often introduced. After reaching a certain recursion depth, the sequential merge sort is used.

Bubble Sort

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

void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

void bubbleSortSeq(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        bool swapped = false;
        for (int j = 0; j < n - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}

void bubbleSortParallel(int arr[], int n) {
    // Perform n iterations of odd-even phases
    for (int i = 0; i < n; i++) {
        // Alternate starting index: 0 for even phase, 1 for odd phase
        int start = i % 2;
        #pragma omp parallel for
        for (int j = start; j < n - 1; j += 2) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
   }}}}

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

    int *seq = new int[n];
    int *par = new int[n];

    cout << "Enter elements: ";
    for (int i = 0; i < n; i++) {
        cin >> seq[i];
        par[i] = seq[i]; // Copy data for parallel sort
    }

    double t1 = omp_get_wtime();
    bubbleSortSeq(seq, n);
    double t2 = omp_get_wtime();
    cout << "Sequential Sorted: ";
    for (int i = 0; i < n; i++)
        cout << seq[i] << " ";
    cout << "\nTime taken (sequential): " << t2 - t1 << " seconds\n";

    t1 = omp_get_wtime();
    bubbleSortParallel(par, n);
    t2 = omp_get_wtime();
    cout << "Parallel Sorted: ";
```

```
    for (int i = 0; i < n; i++)
        cout << par[i] << " ";
    cout << "\nTime taken (parallel): " << t2 - t1 << " seconds\n";

    delete[] seq;
    delete[] par;
    return 0;
}
```

Explanation

1. **Helper Function (`swap`):**
   A simple function to swap two integers.

2. **Sequential Bubble Sort (`bubbleSortSeq`):**
   - Iterates over the array, comparing and swapping adjacent elements if needed.
   - Uses a `swapped` flag to exit early if no swaps occur during an iteration.

3. **Parallel Bubble Sort (`bubbleSortParallel`):**
   - Uses the odd–even transposition method.
   - The starting index for the inner loop alternates between 0 and 1 on each pass.
   - The inner loop is parallelized using OpenMP's `#pragma omp parallel for` directive, so multiple comparisons/swaps occur simultaneously.

4. **`main` Function:**
   - Reads the number of elements and the elements themselves.
   - Duplicates the array to ensure both sorts work on identical data.
   - Measures execution time for both sequential and parallel sorts using `omp_get_wtime()`.

Merge Sort

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

// Merge two sorted subarrays a[left..mid] and a[mid+1..right]
void merge(int a[], int left, int mid, int right) {
    int n = right - left + 1;
    int* temp = new int[n];
    int i = left, j = mid + 1, k = 0;

    while (i <= mid && j <= right) {
        if (a[i] < a[j])
            temp[k++] = a[i++];
        else
            temp[k++] = a[j++];
    }
    while (i <= mid)
        temp[k++] = a[i++];
    while (j <= right)
        temp[k++] = a[j++];

    // Copy back to original array
    for (i = left, k = 0; i <= right; ++i, ++k)
        a[i] = temp[k];

    delete[] temp;
}

// Sequential merge sort
void mergeSortSeq(int a[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSortSeq(a, left, mid);
        mergeSortSeq(a, mid + 1, right);
        merge(a, left, mid, right);
    }
}

// Parallel merge sort using OpenMP sections
void mergeSortPar(int a[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        #pragma omp parallel sections
        {
            #pragma omp section
            mergeSortPar(a, left, mid);
            #pragma omp section
            mergeSortPar(a, mid + 1, right);
        }
        merge(a, left, mid, right);
    }
}

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

    int* seq = new int[n];
    int* par = new int[n];
```

```
    cout << "Enter elements:\n";
    for (int i = 0; i < n; i++) {
        cin >> seq[i];
        par[i] = seq[i]; // Copy data for parallel sort
    }

    // Sequential merge sort
    double start = omp_get_wtime();
    mergeSortSeq(seq, 0, n - 1);
    double end = omp_get_wtime();
    cout << "\nSequential Time: " << end - start << " seconds\n";

    // Parallel merge sort
    start = omp_get_wtime();
    #pragma omp parallel
    {
        #pragma omp single
        mergeSortPar(par, 0, n - 1);
    }
    end = omp_get_wtime();
    cout << "Parallel Time: " << end - start << " seconds\n";

    cout << "\nSorted Array:\n";
    for (int i = 0; i < n; i++)
        cout << par[i] << " ";
    cout << endl;

    delete[] seq;
    delete[] par;

    return 0;
}
```

Explanation

- **Merge Function:**
  Combines two sorted subarrays into one sorted subarray. A temporary array is allocated dynamically to store the merged result, which is then copied back into the original array.
- **Sequential Merge Sort (`mergeSortSeq`):**
  Recursively divides the array until subarrays have one element, then merges them back together.
- **Parallel Merge Sort (`mergeSortPar`):**
  Uses OpenMP's `#pragma omp parallel sections` to sort the left and right halves concurrently. The merge step is performed after the parallel sections.
- **`main` Function:**
  Reads the input array, creates two copies (one for sequential sorting and one for parallel sorting), measures the execution time for each sort using `omp_get_wtime()`, and prints the sorted array along with the time taken.

Assignment 3

Q. Implement Min, Max, Sum, and Average operations using Parallel Reduction

```cpp
#include <iostream>
#include <vector>
#include <omp.h>
#include <climits>
#include <algorithm>  // for std::min and std::max

using namespace std;

void compute_reductions(const vector<int>& arr) {
    int min_value = INT_MAX, max_value = INT_MIN, sum = 0;

    #pragma omp parallel for reduction(min:min_value) reduction(max:max_value)
reduction(+:sum)
    for (size_t i = 0; i < arr.size(); ++i) {
        min_value = min(min_value, arr[i]);
        max_value = max(max_value, arr[i]);
        sum += arr[i];
    }

    double average = static_cast<double>(sum) / arr.size();

    cout << "Minimum: " << min_value << "\n"
         << "Maximum: " << max_value << "\n"
         << "Sum: " << sum << "\n"
         << "Average: " << average << "\n";
}

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

    cout << "Input Array: ";
    for (int val : arr) {
        cout << val << " ";
    }
    cout << "\n\n";

    compute_reductions(arr);

    return 0;
}
```

Explanation

- **Reduction Clauses:**
  The `#pragma omp parallel for` directive splits the loop among threads. The clauses `reduction(min:min_value)`, `reduction(max:max_value)`, and `reduction(+:sum)` create a private copy of each variable for every thread and then combine them after the loop using the specified operations.
- **Using std::min and std::max:**
  Instead of using `if` conditions to update the minimum and maximum values, the code uses `std::min` and `std::max` to make the code cleaner.
- **Computing Average:**
  After obtaining the sum, the average is computed as a double by dividing the sum by the number of elements.

Parallel Reduction – Theory

Parallel Reduction is a technique used in parallel computing to aggregate data (e.g., finding sum, min, max, etc.) from a large array by breaking the task into smaller parts that run concurrently across multiple processors or threads.

How It Works

Instead of performing the operation sequentially on every element (which takes O(n) time), **parallel reduction** performs pairwise operations in **logarithmic time (O(log n))** using multiple threads.

Example Workflow (for Sum):

1. **Start with an array of values**:
   `[a0, a1, a2, a3, ..., an]`
2. **First pass**:
   Each thread adds a pair of elements:
   `[a0 + a1, a2 + a3, ..., an-1 + an]`
3. **Next pass**:
   Threads add results from previous pass:
   `[(a0+a1) + (a2+a3), ...]`
4. Continue until one final result remains.

Operations Explained

1. Sum

Add all elements together using the parallel reduction tree described above.

2. Min

Each thread compares two elements and passes the smaller one. Repeat until the global minimum is found.

3. Max

Same as Min, but pass the **larger** of each pair.

4. Average

First compute the **Sum** using parallel reduction, then divide by total number of elements (can be done in a single thread or as a final parallel step).

Advantages

- Efficient use of CPU/GPU cores
- Reduces time complexity from **O(n)** to **O(log n)**
- Scales well with large datasets

Applications

- Data analytics (mean, median, range)
- Machine learning preprocessing
- Physics and simulation calculations

Assignment 4

Write a CUDA Program for:

1. Vector Addition using CUDA

Theory

CUDA allows you to offload parallel computations onto NVIDIA GPUs. For vector addition:

- **Host Code:** Allocate memory on the host (CPU) and initialize two input vectors.
- **Device Memory Allocation:** Allocate memory on the GPU (device) for the input vectors and the result.
- **Memory Transfer:** Copy the input data from host memory to device memory.
- **Kernel Launch:** Write a CUDA kernel where each thread computes one element of the output vector by adding the corresponding elements from the two input vectors.
- **Result Transfer:** After the kernel finishes, copy the result from the device back to the host.
- **Cleanup:** Free allocated GPU memory.

```
#include <iostream>
#include <cuda.h>
using namespace std;

// CUDA Kernel for vector addition
__global__ void vectorAdd(const float* A, const float* B, float* C, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}

int main() {
    int n = 1 << 20;  // 1M elements
    size_t size = n * sizeof(float);

    // Allocate host memory
    float* h_A = new float[n];
    float* h_B = new float[n];
    float* h_C = new float[n];

    // Initialize input vectors
    for (int i = 0; i < n; i++) {
        h_A[i] = static_cast<float>(i);
        h_B[i] = static_cast<float>(2 * i);
    }

    // Allocate device memory
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    // Copy host data to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

```
    // Launch kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, n);

    // Copy result back to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Verify a few results
    cout << "Sample results:" << endl;
    for (int i = 0; i < 5; i++) {
        cout << h_A[i] << " + " << h_B[i] << " = " << h_C[i] << endl;
    }

    // Clean up
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    delete[] h_A;
    delete[] h_B;
    delete[] h_C;

    return 0;
}
```

2. Matrix Multiplication using CUDA

Theory

For matrix multiplication (computing C = A * B):

- **Host Setup:** Prepare two input matrices (A and B) and an output matrix (C) on the host.
- **Device Memory:** Allocate memory on the device for these matrices.
- **Kernel Design:** Each thread calculates one element of the output matrix C. The thread uses its row and column indices to loop over the shared dimension (number of columns in A or rows in B) to accumulate the dot product.
- **Memory Transfer:** Copy matrices from the host to the device before launching the kernel, and then copy the resulting matrix back.
- **Optimization Note:** While this example uses a straightforward (naive) implementation, real applications often use shared memory tiling for better performance.

```
#include <iostream>
#include <cuda.h>
using namespace std;

#define N 512  // Assuming square matrices of size N x N

// CUDA Kernel for Matrix Multiplication
__global__ void matrixMul(const float* A, const float* B, float* C, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < n && col < n) {
        float sum = 0.0f;
        for (int k = 0; k < n; k++) {
            sum += A[row * n + k] * B[k * n + col];
        }
        C[row * n + col] = sum;
    }
}

int main() {
    int size = N * N * sizeof(float);

    // Allocate host memory for matrices
    float *h_A = new float[N * N];
    float *h_B = new float[N * N];
    float *h_C = new float[N * N];

    // Initialize matrices A and B with some values
    for (int i = 0; i < N * N; i++) {
        h_A[i] = 1.0f;    // You can change these values as needed
        h_B[i] = 2.0f;
    }

    // Allocate device memory
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);
```

```
    // Copy matrices from host to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Define block and grid dimensions
    dim3 threadsPerBlock(16, 16);
    dim3 blocksPerGrid((N + threadsPerBlock.x - 1) / threadsPerBlock.x,
                       (N + threadsPerBlock.y - 1) / threadsPerBlock.y);

    // Launch the matrix multiplication kernel
    matrixMul<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy the result matrix from device to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Display a few elements from the result matrix
    cout << "Sample results from matrix C:" << endl;
    for (int i = 0; i < 5; i++) {
        cout << h_C[i] << " ";  // Print first 5 elements of row 0
    }
    cout << endl;

    // Clean up memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    delete[] h_A;
    delete[] h_B;
    delete[] h_C;

    return 0;
}
```

Summary

- **CUDA Vector Addition:**
  The kernel `vectorAdd` computes the sum for each element. The host code allocates memory, copies data to the GPU, launches the kernel, and then copies the result back.
- **CUDA Matrix Multiplication:**
  The kernel `matrixMul` calculates each element of the output matrix using the dot product of a row from A and a column from B. The host code follows a similar process as in vector addition, setting up device memory, launching the kernel with a 2D grid, and copying the result back.

Compile these examples with a CUDA-capable compiler (like NVCC):

```
nvcc -O2 vector_addition.cu -o vector_addition
nvcc -O2 matrix_mul.cu -o matrix_mul
```

Deep Learning

Assignment 1

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.

1.1 What is Linear Regression?

Linear Regression is a method to **predict a value (output)** based on one or more **input features** using a **straight line**.

Example:

Let’s say you want to predict someone's **salary** based on **years of experience**.

A possible relationship:

```
Salary = 5000 × (Years of Experience) + 20000
```

Here:

- 5000 is the **slope** (how much salary increases per year)
- 20000 is the **intercept** (starting salary)

1.2 Key Idea of Linear Regression:

- Try to draw a **line of best fit** through the data points
- This line is used to make **predictions**

1.3 Example of Linear Regression:

Years of Experience	Salary (in ₹)
1	25,000
2	30,000
3	35,000

We train a model to **learn the pattern** and then predict salary for **4 years** of experience.

1.4 What is a Deep Neural Network (DNN)?

A **Deep Neural Network** is just a **collection of many layers of neurons**, inspired by how the **human brain** works.

Each layer takes input, **processes it**, and passes the result to the next layer.

Think of it like:

```
Input → Hidden Layer 1 → Hidden Layer 2 → ... → Output
```

If there’s only **1 hidden layer**, it’s just a **neural network**.

If there are **2 or more hidden layers**, it’s called **deep**.

1.5 How Does a Deep Neural Network Work?

1. **Input Layer**: Receives the data (e.g. number of rooms, size, location)
2. **Hidden Layers**: Each layer learns **patterns**, **weights**, and **nonlinear relationships**
3. **Output Layer**: Gives final prediction (e.g. price of the house)

Each neuron uses a **mathematical function** (called an **activation function**) to process the inputs.

Learning Process:

- Predict using the current weights
- Compare prediction to actual value
- Calculate **error**
- Use **backpropagation** to adjust weights
- Repeat!

Advantages of Using DNN for Regression

Advantage	Explanation
Can model complex patterns	Understands non-linear relationships
Learns automatically	No need for manual feature engineering
Scales well with data	Works well with many features

Disadvantages

Disadvantage	Explanation
Needs more data	DNNs perform best with large datasets
Slower to train	More layers = more computation
Risk of overfitting	Especially if model is too complex for small data

---

Data

Column Name	Description (in Easy Words)
ID	Just the unique number of each row (not used in prediction)
crim	**Crime rate** — per person in the town. Higher = worse neighborhood, might lower house prices.
zn	**Proportion of residential land** zoned for big plots (e.g., >25,000 sq.ft.). Higher = more spacious homes.
indus	**Proportion of land used for industry** (e.g., factories). Higher = more industrial = less residential.
chas	**Charles River dummy variable**. 1 if the house is **next to the river**, 0 if not. Houses near rivers are usually more expensive.
nox	**Nitric oxide pollution level** (air pollution). Higher = worse air = lower house prices.
rm	**Average number of rooms** per house. More rooms = bigger house = higher price.
age	**Age of the houses** (as % built before 1940). Higher = older homes.
dis	**Distance to employment centers** (downtown). Higher = farther from city = can be cheaper or quieter.
rad	**Accessibility to highways**. Higher = better access to roads.
tax	**Property tax rate** per $10,000. Higher taxes might lower appeal.
ptratio	**Pupil–teacher ratio** in local schools. Lower = better schools = higher house price.
black	A measure related to African-American population (historical/controversial feature).
lstat	**% of low-income population**. Higher = poorer area = lower house price.
medv	**Median value of owner-occupied homes** (in $1000s). This is what we are trying to **predict**!

Code

```
correlation = data.corr()
correlation.loc['medv']
```

This code is used to **analyze how strongly each feature is related to house prices (`medv`)** using **correlation**.

What is Correlation?

- **Correlation** measures how two variables move **together**.
- It gives a value between **-1** and **1**:

Value	Meaning
+1	Perfect **positive** correlation (as one increases, so does the other)
0	No correlation
-1	Perfect **negative** correlation (as one increases, the other decreases)

Feature	Correlation	Meaning
`ID`	-0.22	Not much impact (row index, ignore in modeling)
`crim`	-0.41	Higher crime → lower house prices
`zn`	+0.34	More big residential zones → higher house prices
`indus`	-0.47	More industrial areas → lower prices
`chas`	+0.20	Near the Charles River → slightly higher prices
`nox`	-0.41	More pollution (nitric oxides) → lower prices
`rm`	+0.69	**More rooms → higher price** (Strong positive!)
`age`	-0.36	Older buildings → lower prices
`dis`	+0.25	Farther from employment centers → slightly higher (possibly quieter areas)
`rad`	-0.35	More highway access = may reduce price (maybe noise?)
`tax`	-0.44	Higher taxes → lower price
`ptratio`	-0.48	Higher student-teacher ratio (worse schools) → lower price
`black`	+0.33	Weak positive, but this is a legacy and sensitive feature
`lstat`	-0.74	**Higher % of low-income residents → lower house prices** (Strongest negative correlation)
`medv`	1.0	Perfect correlation with itself

What Matters Most?

Strongest positive impact:

- `rm` (+**0.69**) – More rooms = higher price

Strongest negative impact:

- `lstat` (**–0.74**) – Higher % of low-income residents = lower price
- `ptratio` (**–0.48**) – Worse student-teacher ratio = lower price

Training DNN

Step 5: Normalize (scale) the data

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

What's happening:

- We are **scaling the features** (input values like `RM`, `LSTAT`, etc.) so they are all on the **same scale** — mean = 0 and standard deviation = 1.

Why this is important:

- Deep Neural Networks train **faster and better** when the input data is scaled.
- It prevents some features from **dominating** just because they have larger values (like `TAX` or `AGE`).
- We `fit` the scaler on training data, and `transform` both train and test data to maintain consistency.

Step 6: Build the DNN model

```
model = keras.Sequential([
    keras.layers.Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dense(1)
])
```

What's happening:

- We build a **simple deep neural network** with:
    - 2 hidden layers, each with **64 neurons**
    - Each uses `relu` activation (good for non-linear patterns)
    - Final layer has **1 neuron** with **no activation** because we're predicting a continuous number (house price)

Why this is important:

- This structure helps the model learn complex relationships in the data.
- The last layer outputs a **single value** = predicted house price in $1000s.

Step 7: Compile the model

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

What's happening:

- We're preparing the model for training.

Why this is important:

- `optimizer='adam'`: helps the model adjust weights efficiently.
- `loss='mse'` (Mean Squared Error): calculates how far predictions are from real prices.
- `metrics=['mae']` (Mean Absolute Error): easier to interpret since it's in the same units as the target.

Step 8: Train the model

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

What's happening:

- The model is learning by looking at **training data**.
- We run for **100 rounds (epochs)**.
- `validation_split=0.2` means 20% of training data is used to **check the model's performance** while training.

Why this is important:

- Helps avoid **overfitting** by validating during training.
- We track how the model improves with each epoch.

Step 9: Evaluate the model

```
loss, mae = model.evaluate(X_test_scaled, y_test, verbose=2)
print(f"\nMean Absolute Error on test data: ${mae * 1000:.2f}")
```

What's happening:

- We test the model on **unseen test data** to see how well it performs.
- `mae` (Mean Absolute Error) shows how far off, on average, our predictions are from real values.

Why this is important:

- This is the **final performance check** — how well your model can predict **real-world house prices**.

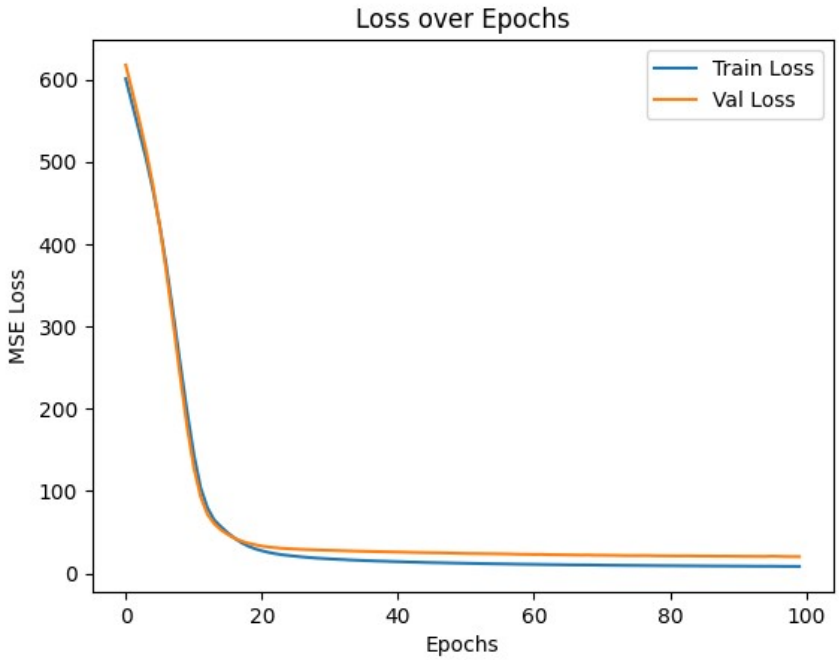
Final Output Example:

```
Mean Absolute Error on test data: $2567.43
```

Means: On average, our predictions are **$2,567 off** from the actual house prices.

Graph

```
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.title('Loss over Epochs')
plt.xlabel('Epochs')
plt.ylabel('MSE Loss')
plt.legend()
plt.show()
```



Example

Epoch	Train Loss	Val Loss	Meaning
1	High	High	Just starting, model not trained
30	Lower	Lower	Model is learning well
60	Low	Going up	Model may be overfitting

Assignment 2

Multiclass classification using Deep Neural Networks: Example: Use the OCR letter recognition dataset https://archive.ics.uci.edu/ml/datasets/letter+recognition

What is Multiclass Classification?

Multiclass classification is a type of machine learning problem where the goal is to classify inputs into **more than two categories** (or classes).

In Simple Terms:

- **Binary classification**: Only two possible outcomes (e.g., Spam or Not Spam).
- **Multiclass classification**: More than two possible outcomes.

Examples of Multiclass Classification:

Input (Features)	Output (Label)
Image of a digit	Predict the number (0–9)
Handwriting data	Predict the letter (A–Z)
Patient symptoms	Predict the disease (flu, cold, allergy, etc.)
News article text	Predict the category (sports, tech, politics, etc.)

OCR Letter Recognition Dataset (UCI)

Here's what the **OCR dataset** does:

- It contains information extracted from handwritten letters (like their shape, pixel count in areas, symmetry, etc.).
- Each data point represents **one letter from A to Z**.
- So, we're trying to **predict which letter** it is.

Dataset: UCI Letter Recognition

Dataset Details

- **Features (Inputs)**: 16 numerical values (e.g., width, height, pixel counts)
- **Target (Output)**: One of the **26 uppercase letters (A to Z)**

So this is a **26-class classification problem**.

How Deep Neural Networks Help

A **Deep Neural Network (DNN)** learns to recognize complex patterns in data and can be used for multiclass classification by:

Model Setup:

- **Input Layer**: Takes in 16 features
- **Hidden Layers**: Learn patterns
- **Output Layer**: Has **26 neurons**, one for each letter (A–Z)
- Uses **softmax activation** in the output layer to predict **probability for each class**

How It Works Step-by-Step:

1. **Input**: Feed the 16 features (from a letter image) into the model.
2. **Processing**: The model passes data through layers, learning patterns.
3. **Output**: The model predicts the most likely letter by choosing the class with the highest probability from the softmax output.

Example:

Let's say we give the model these features:
`[2, 8, 3, 5, 10, 4, ..., 6]` → representing characteristics of a handwritten letter.

The model might output:

```
A: 0.01
B: 0.03
...
E: 0.92 ← Highest
...
Z: 0.001
```

Prediction: **E**

Advantages of Multiclass Classification with DNNs:

- Can learn complex, non-linear relationships.
- Very accurate with enough data and tuning.
- Scales well with large numbers of classes.

Disadvantages:

- Needs more training data than binary classification.
- More chances of **misclassification** between similar classes (e.g., O and D).
- Slower training with larger networks.

Let's break down the code into parts and explain what each section means.

1. Normalization of Image Data

```
x_train = x_train.reshape(60000, 784).astype('float32') / 255
x_test = x_test.reshape(10000, 784).astype('float32') / 255
```

- **`x_train.reshape(60000, 784)`**:
  - The original images in the MNIST dataset are 28x28 pixels.
  - Here, we are reshaping the images into a 1D vector of 784 (28 * 28) elements for each image, as neural networks typically expect 1D arrays (feature vectors) as input.
  - `60000` is the number of training images.
- **`.astype('float32')`**:
  - Converts the pixel values to `float32` (32-bit floating point format), which is more suitable for neural network training.
- **`/ 255`**:
  - Divides the pixel values by 255 to normalize them into the range [0, 1].
  - The original pixel values range from 0 to 255, where 0 is black and 255 is white. By dividing by 255, the pixel values are scaled down to the range [0, 1], which helps the model learn more efficiently.

2. One-Hot Encoding of Labels

```
y_train = np.eye(10)[y_train]
y_test = np.eye(10)[y_test]
```

- **`np.eye(10)`**:
  - Creates a 2D identity matrix of size 10x10. This matrix has 1s on the diagonal and 0s elsewhere.
  - The identity matrix represents the 10 classes (digits from 0 to 9), where each class corresponds to a unique position in the array.
- **`[y_train]`**:
  - This indexing converts each label into a one-hot encoded vector.
  - For example, if `y_train` has the value `3`, `np.eye(10)[3]` will return `[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]`, representing the number `3` as a one-hot vector.
  - This transformation is necessary for multi-class classification, as the model will output probabilities for each class (one of 10 classes in this case).

3. Define the Model Architecture

```
model = Sequential([
    Dense(512, activation='relu', input_shape=(784,)),
    Dropout(0.2),
    Dense(512, activation='relu'),
    Dropout(0.2),
    Dense(10, activation='softmax')
])
```

- **`Sequential()`**:
  - This creates a simple stack of layers, meaning each layer is connected sequentially to the next.
- **`Dense(512, activation='relu', input_shape=(784,))`**:
  - **Dense layer**: This is a fully connected layer where each neuron is connected to every neuron in the previous layer.
  - **`512`**: The number of neurons in this layer. More neurons allow the network to capture more complex patterns.
  - **`activation='relu'`**: The activation function is Rectified Linear Unit (ReLU). It adds non-linearity to the model and is commonly used in hidden layers.
  - **`input_shape=(784,)`**: This specifies that each input has 784 features (because each image is reshaped into a 784-dimensional vector).
- **`Dropout(0.2)`**:
  - This is a regularization technique where randomly selected neurons are ignored during training. The `0.2` means that 20% of the neurons will be dropped (set to zero) during each training step to prevent overfitting.
- **`Dense(10, activation='softmax')`**:
  - This is the output layer. It has 10 neurons (one for each digit 0 to 9).
  - **`activation='softmax'`**: This is the softmax activation function, which converts the raw output scores (logits) into probabilities that sum up to 1. The class with the highest probability is the model's prediction.

4. Compile the Model

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

- **`loss='categorical_crossentropy'`**:
  - This is the loss function used for multi-class classification. It measures how well the predicted probabilities match the true labels.
- **`optimizer=RMSprop()`**:
  - The RMSprop optimizer is used to adjust the weights during training. It is an adaptive learning rate optimization algorithm that helps the model converge faster.

- **`metrics=['accuracy']`**:
    - This specifies the metrics that we want to track during training. In this case, we want to monitor the model's accuracy (the percentage of correct predictions).

5. Train the Model

```
model.fit(x_train, y_train, batch_size=128, epochs=20, validation_data=(x_test, y_test))
```

- **`batch_size=128`**:
    - The number of training samples to be processed before updating the model's weights. A batch size of 128 means that the model will process 128 samples at a time.
    - Smaller batch sizes can lead to better generalization, but larger batches may result in faster training.
- **`epochs=20`**:
    - The number of times the model will iterate over the entire training dataset.
    - Each epoch means one complete pass through the dataset, and after each epoch, the model weights are updated based on the loss and optimizer.
- **`validation_data=(x_test, y_test)`**:
    - This specifies the validation data that the model will use to evaluate its performance after each epoch. The model will compute the loss and accuracy on the `x_test` and `y_test` data during training to monitor its performance on unseen data.

Key Concepts:

- **Batch Size**: The number of training examples used in one update of the model's weights. A larger batch size can speed up training but might hurt model generalization.
- **Epochs**: The number of times the entire training dataset is passed through the model. More epochs typically lead to better performance (up to a point) but also increases the risk of overfitting.

Assignment 3

Use MNIST Fashion Dataset and create a classifier to classify fashion clothing into categories.

1. What is Classification?

Classification is a type of machine learning task where the goal is to assign labels (categories) to input data based on its features. Imagine you have pictures of clothes, and your job is to tell whether each picture shows a shirt, pants, or shoes. In other words, you're trying to figure out which "category" each item belongs to. In classification, you train a machine to recognize patterns in the data and assign it to the correct category.

For example:

- If you show a picture of a T-shirt, the model should classify it as "T-shirt".
- If you show a picture of pants, it should classify it as "Pants".

2. What is CNN (Convolutional Neural Network)?

A **Convolutional Neural Network (CNN)** is a special type of deep learning model used for processing images. It is a type of neural network designed specifically to recognize patterns, shapes, textures, and other important features in images.

In a CNN, there are different layers that help it understand images:

- **Convolutional Layers**: These layers look for small patterns (like edges, shapes, or textures) in images.
- **Pooling Layers**: These layers reduce the size of the image data while keeping important features.
- **Fully Connected Layers**: These layers make final decisions about what the image is based on the patterns found in the previous layers.

CNNs are great for image-related tasks because they automatically learn and extract useful features from images, rather than needing you to manually define the patterns.

3. How Do Deep Neural Networks Work on Classification Using CNN?

A **Deep Neural Network (DNN)** is a machine learning model made of layers of neurons (small computational units). The network learns by adjusting the weights of these neurons during training, trying to minimize the error in predictions. When combined with CNN, it works as follows:

- **Step 1: Input** - You provide the model with an image (like a picture of a T-shirt).
- **Step 2: Convolution** - The CNN scans the image using small filters to look for simple patterns (like edges or shapes).
- **Step 3: Pooling** - The network reduces the size of the image while retaining the important patterns found in the first step.
- **Step 4: Flattening** - The data is flattened into a 1D vector (a long list) to be processed by fully connected layers.

- **Step 5: Fully Connected Layers** - The neural network then uses the information from the previous steps to make a decision on what the image represents.
- **Step 6: Output** - The network produces a prediction, such as "T-shirt" or "Pants."

Why CNN for Fashion Classification?

CNNs are particularly good for classifying fashion items because they can focus on small regions of the image and capture essential patterns that are important for distinguishing between items of clothing. For example, CNN might recognize the shape of sleeves or the texture of pants, which helps it decide whether an item is a T-shirt or a pair of jeans.

4. What is the MNIST Fashion Dataset?

The **MNIST Fashion dataset** is a collection of 60,000 images of fashion items that are labeled into 10 categories, such as:

- T-shirts
- Pants
- Dresses
- Sneakers
- Sandals
- Boots
- And more

Each image is a grayscale picture of clothing that is 28x28 pixels in size. This dataset is widely used for teaching machine learning and deep learning models because it is simple enough for beginners and still provides useful insights into how classifiers work.

The categories in the MNIST Fashion dataset are:

1. T-shirt/top
2. Trouser
3. Pullover
4. Dress
5. Coat
6. Sandal
7. Shirt
8. Sneaker
9. Bag
10. Ankle boot

You use this dataset to train your model, so it can classify unseen fashion items into one of these categories.

The Process of Building a Fashion Classifier Using CNN:

To create a model that can classify fashion items into categories, you generally follow these steps:

1. **Load the Data**: You load the MNIST Fashion dataset (images and labels).
2. **Preprocess the Data**: You normalize the images so that pixel values are between 0 and 1, which helps the model learn better.
3. **Create a CNN Model**: You define the architecture of the CNN, which involves:
    - Convolutional layers (to extract patterns from images),
    - Pooling layers (to reduce image size while keeping important features),
    - Fully connected layers (to classify the image into one of the categories).
4. **Train the Model**: You feed the preprocessed images and labels into the model, and it learns to classify the images.
5. **Evaluate the Model**: After training, you test the model on new images to see how accurately it can predict the category of unseen images.
6. **Make Predictions**: You can then use the trained model to classify new fashion images.

1. Normalize the Pixel Values to be Between 0 and 1

```
x_train = x_train / 255.0
x_test = x_test / 255.0
```

- **What is Normalization?**
    - In the original MNIST dataset (and many other datasets), pixel values are integers ranging from 0 to 255, where 0 represents black and 255 represents white (in grayscale images).
    - **Normalization** is a technique to scale the data so that values are between 0 and 1. This helps neural networks learn faster and more effectively.
- **How does this work?**
    - Dividing the pixel values by 255 converts them from an integer range (0-255) to a floating-point range (0.0-1.0). For example, if a pixel value is 255, after division, it becomes 1. If the pixel value is 0, it remains 0.

2. Reshape the Data (28x28 Images to a Flat 1D Vector of 784 Values)

```
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)
```

- **What does Reshaping Mean?**
    - Each image in the MNIST Fashion dataset is a **28x28 pixel** grayscale image, which means it has 28 rows and 28 columns. Each pixel in the image has one value (for grayscale images).

- The neural network we're building expects data in a specific format. For Convolutional Neural Networks (CNNs), the data is expected in 4D form: `[samples, height, width, channels]`.

- **How does this work?**
  - `.reshape(-1, 28, 28, 1)` reshapes the `x_train` and `x_test` datasets:
    - `-1` means "let it be automatically inferred from the other dimensions" (in this case, it's the number of images).
    - `28, 28` refers to the image dimensions (height and width).
    - `1` refers to the number of color channels (1 for grayscale, 3 for RGB).

So, this reshapes the images from a flat 28x28 array of pixel values to a format suitable for CNN processing.

3. Define the Model Architecture Using CNN

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

model = Sequential([
    Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(64, kernel_size=(3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')  # 10 classes (fashion categories)
])
```

- **What is a CNN (Convolutional Neural Network)?**
  - CNNs are a type of deep learning model designed for processing data that has a grid-like structure, such as images.
  - CNNs apply convolutional layers to detect patterns and features in images, such as edges, shapes, and textures.

- **Model Breakdown:**
  - **Conv2D Layer:**
    - This is a **Convolutional layer** that applies a 2D filter (or kernel) to the input image.
    - `32`: Number of filters used.
    - `kernel_size=(3, 3)`: Size of the filter (3x3 pixels).
    - `activation='relu'`: ReLU (Rectified Linear Unit) is a popular activation function that helps introduce non-linearity into the network.
    - `input_shape=(28, 28, 1)`: The input shape of each image (28x28 pixels, 1 channel for grayscale).

- **MaxPooling2D Layer:**
  - This is a **Pooling layer** that reduces the size of the image by downsampling.
  - `pool_size=(2, 2)`: A 2x2 window is used to take the maximum value from the 2x2 region of the image. This helps reduce computation and prevent overfitting.
- **Another Conv2D and MaxPooling2D Layer:**
  - The second pair of convolutional and pooling layers allows the network to learn more complex features from the image.
- **Flatten Layer:**
  - This layer **flattens** the 2D feature map into a 1D vector to connect it to fully connected layers.
- **Dense Layer:**
  - A fully connected layer (Dense) that takes the flattened data and passes it through.
  - `128` neurons are used in this layer.
  - `activation='relu'`: ReLU is again used here.
- **Output Layer:**
  - `Dense(10, activation='softmax')`: This is the output layer with 10 neurons (one for each class/category).
  - `softmax`: This activation function is used for multi-class classification problems. It ensures that the output is a probability distribution across the 10 classes.

4. Compile the Model

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

- **Compile Process:**
  - **optimizer='adam'**: The Adam optimizer is an efficient optimization algorithm that adjusts the learning rate during training.
  - **loss='sparse_categorical_crossentropy'**: This is the loss function used for multi-class classification. It measures how far the predicted class probabilities are from the actual labels.
  - **metrics=['accuracy']**: The metric to monitor during training is **accuracy** (how often the model's predictions are correct).

5. Train the Model

```
model.fit(x_train, y_train, epochs=5)
```

- **Training the Model:**
  - **x_train** and **y_train** are the training data (images and labels).
  - **epochs=5**: This means the model will go through the entire training dataset 5 times to learn from it.
  - The model adjusts its weights during each pass to minimize the loss function.

6. Evaluate the Model on Test Data

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

- **Evaluate the Model:**
  - `x_test` and `y_test` are the test data and labels that the model has never seen before.
  - `model.evaluate()` calculates the loss and accuracy on the test data to see how well the model generalizes to new, unseen data.
  - The test accuracy is printed, showing how well the model performs.