BFS

```
#include <iostream>             // For input/output operations
#include <vector>              // For using vector container
#include <queue>               // For using queue data structure
#include <omp.h>               // For OpenMP functions and parallelization

using namespace std;

// Function to perform parallel BFS traversal
void parallelBFS(const vector<vector<int>> &graph, int startNode) {
    int n = graph.size();              // Number of nodes in the graph
    vector<bool> visited(n, false);     // Track visited nodes
    queue<int> q;                      // Queue for BFS traversal

    visited[startNode] = true;          // Mark starting node as visited
    q.push(startNode);                 // Enqueue starting node

    cout << "BFS Traversal: ";

    // Main BFS loop
    while (!q.empty()) {
        int size = q.size();           // Current level size
        vector<int> next_level;        // Store next-level nodes to enqueue

        // Parallel loop to process all nodes at current BFS level
        #pragma omp parallel for shared(q, visited, graph) default(none) firstprivate(size)
schedule(dynamic)
        for (int i = 0; i < size; ++i) {
            int current;

            // Critical section to safely pop from queue
            #pragma omp critical
            {
                if (!q.empty()) {
                    current = q.front();    // Get front node
                    q.pop();                // Remove it from queue
                    cout << current << " "; // Print the node
                }
            }

            // Visit all neighbors of current node
            for (int neighbor : graph[current]) {
                // Critical section to check and mark visited
                #pragma omp critical
                {
                    if (!visited[neighbor]) {
                        visited[neighbor] = true;      // Mark as visited
```

```cpp
                        next_level.push_back(neighbor); // Queue for next level
                    }
                }
            }
        }

        // Push next-level nodes into the queue
        for (int node : next_level) {
            q.push(node);
        }
    }

    cout << endl;
}

int main() {
    // Define a graph using adjacency list
    vector<vector<int>> graph = {
        {1, 2},     // Node 0 is connected to 1 and 2
        {0, 3, 4},  // Node 1 is connected to 0, 3, 4
        {0, 5, 6},  // Node 2 is connected to 0, 5, 6
        {1},        // Node 3 is connected to 1
        {1},        // Node 4 is connected to 1
        {2},        // Node 5 is connected to 2
        {2}         // Node 6 is connected to 2
    };

    int startNode = 0;  // Start BFS from node 0

    // Measure time using OpenMP wall-clock timer
    double start = omp_get_wtime();    // Start time
    parallelBFS(graph, startNode);     // Perform parallel BFS
    double end = omp_get_wtime();      // End time

    // Print execution time
    cout << "Execution Time: " << end - start << " seconds" << endl;

    return 0;
}
```

```
g++ -fopenmp -o parallel_bfs parallel_bfs.cpp
./parallel_bfs
```

DFS

```
#include <iostream>             // For standard input and output
#include <vector>               // For using vector container (dynamic array)
#include <stack>                // For using stack data structure (DFS uses stack)
#include <omp.h>                // For using OpenMP functions for parallelism

using namespace std;

// Function to perform parallel DFS
void parallelDFS(const vector<vector<int>> &graph, int startNode) {
    int n = graph.size();                   // Number of nodes in the graph
    vector<bool> visited(n, false);         // To track visited nodes
    stack<int> s;                           // Stack to implement DFS

    s.push(startNode);                      // Push the starting node onto the stack

    cout << "DFS Traversal: ";

    // Start parallel region
    #pragma omp parallel
    {
        while (true) {
            int current;

            // Use critical section to safely access shared stack
            #pragma omp critical
            {
                if (!s.empty()) {
                    current = s.top();      // Get the top node from the stack
                    s.pop();                // Remove the node from the stack
                } else {
                    current = -1;           // Signal that stack is empty
                }
            }

            if (current == -1)              // Exit condition: stack was empty
                break;

            if (!visited[current]) {
                // Only one thread should mark and print to avoid duplicates
                #pragma omp critical
                {
                    visited[current] = true; // Mark the node as visited
                    cout << current << " ";  // Print the node
                }

                // Traverse neighbors of the current node
```

```
                #pragma omp critical
                {
                    for (int i = graph[current].size() - 1; i >= 0; --i) {
                        int neighbor = graph[current][i];
                        if (!visited[neighbor]) {
                            s.push(neighbor); // Push unvisited neighbors onto the stack
                        }
                    }
                }
            }
        }
    }

    cout << endl;
}

int main() {
    int nodes, edges;
    cout << "Enter number of nodes: ";
    cin >> nodes;
    cout << "Enter number of edges: ";
    cin >> edges;

    // Initialize graph with given number of nodes
    vector<vector<int>> graph(nodes);

    cout << "Enter edges (format: u v means an edge between u and v):" << endl;
    for (int i = 0; i < edges; ++i) {
        int u, v;
        cin >> u >> v;

        // Assuming undirected graph
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    int startNode;
    cout << "Enter starting node for DFS: ";
    cin >> startNode;

    double start = omp_get_wtime();          // Start timer
    parallelDFS(graph, startNode);           // Run DFS
    double end = omp_get_wtime();            // End timer

    cout << "Execution Time: " << end - start << " seconds" << endl;

    return 0;
}
```

Bubble Sort

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

using namespace std;

// Sequential Bubble Sort
void bubbleSortSequential(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; ++i)
        for (int j = 0; j < n - i - 1; ++j)
            if (arr[j] > arr[j + 1])
                swap(arr[j], arr[j + 1]);
}

// Parallel Bubble Sort (Odd-Even Transposition Sort)
void bubbleSortParallel(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n; ++i) {
        #pragma omp parallel for
        for (int j = (i % 2); j < n - 1; j += 2) {
            if (arr[j] > arr[j + 1])
                swap(arr[j], arr[j + 1]);
        }
    }
}

// Print full array
void printArray(const vector<int>& arr) {
    for (int val : arr)
        cout << val << " ";
    cout << endl;
}

int main() {
    int n;
    cout << "Enter size of array: ";
    cin >> n;

    vector<int> inputArray(n);
    cout << "Enter " << n << " elements:\n";
    for (int i = 0; i < n; ++i)
        cin >> inputArray[i];
```

```
    // Copy for both versions
    vector<int> seqArr = inputArray;
    vector<int> parArr = inputArray;

    // Sequential sort
    double start = omp_get_wtime();
    bubbleSortSequential(seqArr);
    double end = omp_get_wtime();
    cout << "\nSequential Bubble Sort Result:\n";
    printArray(seqArr);
    cout << "Time taken (Sequential): " << end - start << " seconds\n";

    // Parallel sort
    start = omp_get_wtime();
    bubbleSortParallel(parArr);
    end = omp_get_wtime();
    cout << "\nParallel Bubble Sort Result:\n";
    printArray(parArr);
    cout << "Time taken (Parallel): " << end - start << " seconds\n";

    return 0;
}
```

Merge Sort

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

using namespace std;

// Merge function to combine two sorted halves
void merge(vector<int>& arr, int left, int mid, int right) {
    vector<int> temp(right - left + 1);
    int i = left, j = mid + 1, k = 0;

    while (i <= mid && j <= right)
        temp[k++] = (arr[i] <= arr[j]) ? arr[i++] : arr[j++];

    while (i <= mid) temp[k++] = arr[i++];
    while (j <= right) temp[k++] = arr[j++];

    for (int x = 0; x < k; ++x)
        arr[left + x] = temp[x];
}

// Sequential Merge Sort
void mergeSortSequential(vector<int>& arr, int left, int right) {
    if (left >= right) return;
    int mid = (left + right) / 2;
    mergeSortSequential(arr, left, mid);
    mergeSortSequential(arr, mid + 1, right);
    merge(arr, left, mid, right);
}

// Parallel Merge Sort using OpenMP
void mergeSortParallel(vector<int>& arr, int left, int right, int depth = 0) {
    if (left >= right) return;
    int mid = (left + right) / 2;

    if (depth < 3) {  // Limit depth of parallel recursion
        #pragma omp parallel sections
        {
            #pragma omp section
            mergeSortParallel(arr, left, mid, depth + 1);

            #pragma omp section
            mergeSortParallel(arr, mid + 1, right, depth + 1);
        }
    } else {
        mergeSortSequential(arr, left, mid);
```

```
        mergeSortSequential(arr, mid + 1, right);
    }

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

// Print full array
void printArray(const vector<int>& arr) {
    for (int val : arr)
        cout << val << " ";
    cout << endl;
}

int main() {
    int n;
    cout << "Enter size of array: ";
    cin >> n;

    vector<int> inputArray(n);
    cout << "Enter " << n << " elements:\n";
    for (int i = 0; i < n; ++i)
        cin >> inputArray[i];

    // Make two copies
    vector<int> seqArr = inputArray;
    vector<int> parArr = inputArray;

    // Sequential Merge Sort
    double start = omp_get_wtime();
    mergeSortSequential(seqArr, 0, n - 1);
    double end = omp_get_wtime();
    cout << "\nSequential Merge Sort Result:\n";
    printArray(seqArr);
    cout << "Time taken (Sequential): " << end - start << " seconds\n";

    // Parallel Merge Sort
    start = omp_get_wtime();
    mergeSortParallel(parArr, 0, n - 1);
    end = omp_get_wtime();
    cout << "\nParallel Merge Sort Result:\n";
    printArray(parArr);
    cout << "Time taken (Parallel): " << end - start << " seconds\n";

    return 0;
}
```

Parallel Reduction

```
#include <iostream>
#include <vector>
#include <omp.h>
#include <climits>  // For INT_MIN, INT_MAX

using namespace std;

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

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

    int minVal = INT_MAX;
    int maxVal = INT_MIN;
    long long sum = 0;

    double start = omp_get_wtime();

    #pragma omp parallel for reduction(min:minVal) reduction(max:maxVal) reduction(+:sum)
    for (int i = 0; i < n; ++i) {
        if (arr[i] < minVal)
            minVal = arr[i];
        if (arr[i] > maxVal)
            maxVal = arr[i];
        sum += arr[i];
    }

    double end = omp_get_wtime();

    double average = (double)sum / n;

    cout << "\nResults using Parallel Reduction:\n";
    cout << "Minimum: " << minVal << endl;
    cout << "Maximum: " << maxVal << endl;
    cout << "Sum: " << sum << endl;
    cout << "Average: " << average << endl;
    cout << "Time taken: " << end - start << " seconds\n";

    return 0;
}
```

Cuda Addition

File - vector_add.cu

```
// File: vector_addition.cu
#include <iostream>
#include <cuda.h>

#define N 1000000  // Size of vectors

// CUDA kernel for vector addition
__global__ void vectorAdd(int* A, int* B, int* C) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;  // Global thread index
    if (i < N)
        C[i] = A[i] + B[i];
}

int main() {
    int *h_A, *h_B, *h_C;        // Host arrays
    int *d_A, *d_B, *d_C;        // Device arrays
    int size = N * sizeof(int);

    // Allocate memory on host
    h_A = (int*)malloc(size);
    h_B = (int*)malloc(size);
    h_C = (int*)malloc(size);

    // Initialize host arrays
    for (int i = 0; i < N; i++) {
        h_A[i] = i;
        h_B[i] = 2 * i;
    }

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

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

    // Launch kernel with enough blocks
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result back to host
```

```
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Display first 10 results
    std::cout << "First 10 elements of vector C (A + B):\n";
    for (int i = 0; i < 10; ++i)
        std::cout << h_C[i] << " ";
    std::cout << std::endl;

    // Free memory
    free(h_A); free(h_B); free(h_C);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);

    return 0;
}
```

How to compile and Run

```
nvcc vector_add.cu -o vector_add
./vector_add

nvcc matrix_mul.cu -o matrix_mul
./matrix_mul
```

Cuda Multi

```
#include <iostream>
#include <cuda.h>

#define N 16  // Size of square matrices

using namespace std;

// CUDA kernel for matrix multiplication
__global__ void matrixMul(int *a, int *b, int *c, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y; // Row index
    int col = blockIdx.x * blockDim.x + threadIdx.x; // Column index

    if (row < n && col < n) {
        int sum = 0;
        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(int);

    // Host memory
    int *h_a = new int[N * N];
    int *h_b = new int[N * N];
    int *h_c = new int[N * N];

    // Initialize matrices
    for (int i = 0; i < N * N; ++i) {
        h_a[i] = 1;
        h_b[i] = 2;
    }

    // Device memory
    int *d_a, *d_b, *d_c;
    cudaMalloc(&d_a, size);
    cudaMalloc(&d_b, size);
    cudaMalloc(&d_c, size);
```

```
    // Copy input matrices to device
    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

    // Launch kernel (1 block per 16 threads in both x and y)
    dim3 threadsPerBlock(16, 16);
    dim3 blocksPerGrid((N + 15) / 16, (N + 15) / 16);
    matrixMul<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, N);

    // Copy result back
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    // Display part of result
    cout << "Result matrix (first 4x4 block):\n";
    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 4; ++j)
            cout << h_c[i * N + j] << " ";
        cout << endl;
    }

    // Free memory
    delete[] h_a;
    delete[] h_b;
    delete[] h_c;
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

Dl 1

To implement **Linear Regression using a Deep Neural Network (DNN)** for the **Boston Housing Price Prediction** problem, we can use **TensorFlow** or **PyTorch**. Below is a full implementation using **TensorFlow (Keras API)**.

Overview

- **Dataset**: Boston Housing (available in `keras.datasets`)
- **Model**: Simple Neural Network (no hidden layers for pure linear regression)
- **Loss**: Mean Squared Error (MSE)
- **Optimizer**: Adam
- **Metrics**: Mean Absolute Error (MAE)

Python Code using TensorFlow/Keras

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import boston_housing
import matplotlib.pyplot as plt

# Load the dataset
(x_train, y_train), (x_test, y_test) = boston_housing.load_data()

# Normalize features
mean = x_train.mean(axis=0)
std = x_train.std(axis=0)
x_train = (x_train - mean) / std
x_test = (x_test - mean) / std

# Linear Regression using DNN (no hidden layers)
model = models.Sequential([
    layers.Dense(1, input_shape=(x_train.shape[1],))  # Linear regression
])

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

```
# Train the model
history = model.fit(x_train, y_train, epochs=100, batch_size=32, validation_split=0.2,
verbose=0)

# Evaluate the model
loss, mae = model.evaluate(x_test, y_test, verbose=1)
print(f"\nTest MAE: {mae:.2f}")

# Predict example
predictions = model.predict(x_test[:5])
print("\nSample Predictions vs Actual:")
for i in range(5):
    print(f"Predicted: {predictions[i][0]:.2f}, Actual: {y_test[i]}")

# Plot training loss
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel("Epoch")
plt.ylabel("Loss (MSE)")
plt.legend()
plt.title("Training vs Validation Loss")
plt.show()
```

Output Example

```
Test MAE: 2.46

Sample Predictions vs Actual:
Predicted: 18.75, Actual: 20.6
Predicted: 19.90, Actual: 19.5
...
```

Requirements

Install TensorFlow:

```
pip install tensorflow matplotlib
```

Dl2

Here are the **fully compatible Jupyter Notebook versions** of both deep neural network classification codes:

---

1. Multiclass Classification: OCR Letter Recognition (Jupyter Version)

Step 1: Download the dataset

- Download `letter-recognition.data` from:

  https://archive.ics.uci.edu/ml/machine-learning-databases/letter-recognition/letter-recognition.data

- Place the file in the **same folder** as your Jupyter notebook.

Code:

```
# %matplotlib inline for plots (if needed later)
%matplotlib inline

import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt

# Load dataset (make sure the file is in the same directory)
columns = ['letter'] + [f'feature_{i}' for i in range(1, 17)]
df = pd.read_csv('letter-recognition.data', header=None, names=columns)

# Split features and labels
X = df.drop('letter', axis=1).values
y = df['letter'].values

# Encode class labels to integers
encoder = LabelEncoder()
y_encoded = encoder.fit_transform(y)
```

```
# One-hot encode the labels
y_onehot = to_categorical(y_encoded)

# Normalize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_onehot, test_size=0.2,
random_state=42)

# Build the model
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=(16,)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(26, activation='softmax')  # 26 classes (A-Z)
])

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

# Train
history = model.fit(X_train, y_train, epochs=20, batch_size=32, validation_split=0.2,
verbose=1)

# Evaluate
loss, accuracy = model.evaluate(X_test, y_test)
print(f"\nTest Accuracy: {accuracy:.2f}")

# Optional: plot training history
plt.plot(history.history['accuracy'], label='Train Acc')
plt.plot(history.history['val_accuracy'], label='Val Acc')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

---

2. Binary Classification: IMDB Sentiment Analysis (Jupyter Version)

Code:

```
import tensorflow as tf
```

```
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Dense, Flatten
import matplotlib.pyplot as plt

# Load dataset
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=10000)

# Pad sequences for equal length input
x_train = pad_sequences(x_train, maxlen=200)
x_test = pad_sequences(x_test, maxlen=200)

# Build the model
model = Sequential([
    Embedding(input_dim=10000, output_dim=32, input_length=200),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid')  # Binary output
])

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

# Train
history = model.fit(x_train, y_train, epochs=5, batch_size=64, validation_split=0.2,
verbose=1)

# Evaluate
loss, acc = model.evaluate(x_test, y_test)
print(f"\nTest Accuracy: {acc:.2f}")

# Plot accuracy
plt.plot(history.history['accuracy'], label='Train Acc')
plt.plot(history.history['val_accuracy'], label='Val Acc')
plt.title('IMDB Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

To Run These in Jupyter Notebook:

- Install requirements if needed:
  pip install tensorflow scikit-learn pandas matplotlib

Dl3

Here are **two separate CNN implementations** compatible with **Jupyter Notebook**, using TensorFlow/Keras:

---

1. Plant Disease Detection using CNN

Dataset: [PlantVillage Dataset]()
You can download and extract it into a `plant_disease/` directory.

Code:

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt

# Set paths (adjust if different)
train_path = 'plant_disease/train'
val_path = 'plant_disease/val'

# Data augmentation and normalization
train_datagen = ImageDataGenerator(rescale=1./255, rotation_range=20, zoom_range=0.2,
horizontal_flip=True)
val_datagen = ImageDataGenerator(rescale=1./255)

# Load data
train_data = train_datagen.flow_from_directory(train_path, target_size=(128, 128),
batch_size=32, class_mode='categorical')
val_data = val_datagen.flow_from_directory(val_path, target_size=(128, 128),
batch_size=32, class_mode='categorical')

# Build CNN model
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(128, 128, 3)),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(train_data.num_classes, activation='softmax')
])
```

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

# Train
history = model.fit(train_data, epochs=10, validation_data=val_data)

# Plot accuracy
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Val Accuracy')
plt.legend()
plt.title('Plant Disease Detection Accuracy')
plt.show()
```

Note: Use `!pip install kaggle` and upload the dataset to Google Colab or download manually.

---

2. CNN on Fashion MNIST Dataset

Code:

```
import tensorflow as tf
import matplotlib.pyplot as plt

# Load dataset
fashion_mnist = tf.keras.datasets.fashion_mnist
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

# Normalize and reshape for CNN input
x_train = x_train / 255.0
x_test = x_test / 255.0
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)

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

```
# Compile and train
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
history = model.fit(x_train, y_train, epochs=5, validation_split=0.2)

# Evaluate
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"\nTest Accuracy: {test_acc:.2f}")

# Plot accuracy
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Val Accuracy')
plt.legend()
plt.title('Fashion MNIST Classification Accuracy')
plt.show()
```

Requirements

pip install tensorflow matplotlib

Would you like confusion matrix or prediction visualization included in these notebooks?

DI4