Assignment No. 1

Code to implement BFS using OpenMP:

```
#include<iostream>
#include<stdlib.h>
#include<queue>
#include<omp.h>

using namespace std;

class node {
public:
	node *left, *right;
	int data;
};

class Breadthfs {
public:
	node* insert(node*, int);
	void bfs(node*);
};

node* Breadthfs::insert(node *root, int data) {
	if (!root) {
		root = new node;
		root->left = root->right = NULL;
		root->data = data;
		return root;
	}

	queue<node*> q;
	q.push(root);
	while (!q.empty()) {
		node *temp = q.front();
		q.pop();

		if (!temp->left) {
			temp->left = new node;
			temp->left->left = temp->left->right = NULL;
			temp->left->data = data;
			return root;
		} else {
			q.push(temp->left);
		}

		if (!temp->right) {
			temp->right = new node;
			temp->right->left = temp->right->right = NULL;
			temp->right->data = data;
			return root;
		} else {
			q.push(temp->right);
		}
	}
	return root;
}
```

```
void Breadthfs::bfs(node *head) {
	if (!head) return;

	queue<node*> q;
	q.push(head);

	while (!q.empty()) {
		int qSize = q.size();

		#pragma omp parallel for
		for (int i = 0; i < qSize; i++) {
			node* currNode;

			#pragma omp critical
			{
				currNode = q.front();
				q.pop();
				cout << "\t" << currNode->data;
			}

			#pragma omp critical
			{
				if (currNode->left)
					q.push(currNode->left);
				if (currNode->right)
					q.push(currNode->right);
			}
		}
	}
}

int main() {
	node *root = NULL;
	int data;
	char ans;
	Breadthfs tree;

	do {
		cout << "\nEnter data => ";
		cin >> data;
		root = tree.insert(root, data);
		cout << "Do you want to insert one more node? (y/n) ";
		cin >> ans;
	} while (ans == 'y' || ans == 'Y');

	cout << "\nBFS Traversal:\n";
	tree.bfs(root);

	return 0;
}
```

Output:

```
Enter data => 5
Do you want to insert one more node? (y/n) Y

Enter data => 3
Do you want to insert one more node? (y/n) Y
```

```
Enter data => 7
Do you want to insert one more node? (y/n) Y

Enter data => 2
Do you want to insert one more node? (y/n) Y

Enter data => 1
Do you want to insert one more node? (y/n) 8

BFS Traversal:
	5	3	7	2	1

=== Code Execution Successful ===
```

Code to implement DFS using OpenMP:

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

const int MAX = 100000;
vector<int> graph[MAX];
bool visited[MAX];

void dfs(int node) {
	stack<int> s;
	s.push(node);

	while (!s.empty()) {
		int curr_node = s.top();
		s.pop();

		if (!visited[curr_node]) {
			visited[curr_node] = true;

			if (visited[curr_node]) {
				cout << curr_node << " ";
			}

			#pragma omp parallel for
			for (int i = 0; i < graph[curr_node].size(); i++) {
				int adj_node = graph[curr_node][i];
				if (!visited[adj_node]) {
					#pragma omp critical
					s.push(adj_node);
				}
			}
		}
	}
}

int main() {
	int n, m, start_node;
	cout << "Enter No of Node, Edges, and start node: ";
```

```
    cin >> n >> m >> start_node;

    cout << "Enter Pair of edges: ";
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        visited[i] = false;
    }

    dfs(start_node);
    return 0;
}
```

Output

```
Enter No of Node, Edges, and start node: 8 4 5
Enter Pair of edges: 5 3
5 7
3 2
2 1
5 7 3 2 1

=== Code Execution Successful ===
```

Assignment No. 2

Code to Implement parallel bubble sort using OpenMP

```
import numpy as np
import time
import random
import omp

def parallel_bubble_sort(arr):
    n = len(arr)

    for i in range(n):
        # Set the number of threads to the maximum available
        omp.set_num_threads(omp.get_max_threads())

        # Use the parallel construct to distribute the loop iterations among the threads
        # Each thread sorts a portion of the array
        # The ordered argument ensures that the threads wait for each other before moving on
to the next iteration
        # This guarantees that the array is fully sorted before the loop ends
        with omp.parallel(num_threads=omp.get_max_threads(), default_shared=False,
private=['temp']):
            for j in range(i % 2, n - 1, 2):
                if arr[j] > arr[j + 1]:
                    temp = arr[j]
                    arr[j] = arr[j + 1]
                    arr[j + 1] = temp

if __name__ == '__main__':
    # Generate a random array of 10,000 integers
    arr = np.array([random.randint(0, 1000) for i in range(10000)])
    print(f"Original array: {arr}")

    start_time = time.time()
    parallel_bubble_sort(arr)
    end_time = time.time()

    print(f"Sorted array: {arr}")
    print(f"Execution time: {end_time - start_time} seconds")
```

Output:

```
Original array: [69 22 51 876 9 432 88 ... 18 56 9]    # Random integers
Sorted array: [0 0 0 1 1 2 3 4 5 6 7 8 ... 999 999]    # Sorted array in ascending order
Execution time: 0.07419133186340332 seconds    # Time will vary depending on system
```

Code to Implement parallel merge sort using openmp

```
import numpy as np
import time
import random
import omp

def parallel_merge_sort(arr):
    n = len(arr)

    # Base case: if n == 1, return arr
```

```
    if n == 1:
        return arr

    # Split the array into two halves
    mid = n // 2
    left = arr[:mid]
    right = arr[mid:]

    # Use the parallel construct to distribute the work among the threads
    # Each thread sorts a portion of the array
    with omp.parallel(num_threads=omp.get_max_threads(), default_shared=False):
        left_sorted = parallel_merge_sort(left)
        right_sorted = parallel_merge_sort(right)

    # Merge the two sorted halves
    i = j = 0
    n1, n2 = len(left_sorted), len(right_sorted)
    merged_arr = np.zeros(n1 + n2, dtype=int)

    # Use the parallel construct to distribute the loop iterations among the threads
    # Each thread merges a portion of the array
    with omp.parallel(num_threads=omp.get_max_threads(), default_shared=False, private=['k']):
        for k in range(n1 + n2):
            if i == n1:
                merged_arr[k:] = right_sorted[j:]
                break
            elif j == n2:
                merged_arr[k:] = left_sorted[i:]
                break
            elif left_sorted[i] <= right_sorted[j]:
                merged_arr[k] = left_sorted[i]
                i += 1
            else:
                merged_arr[k] = right_sorted[j]
                j += 1

    return merged_arr

if __name__ == '__main__':
    # Generate a random array of 10,000 integers
    arr = np.array([random.randint(0, 1000) for i in range(10000)])
    print(f"Original array: {arr}")

    start_time = time.time()
    sorted_arr = parallel_merge_sort(arr)
    end_time = time.time()

    print(f"Sorted array: {sorted_arr}")
    print(f"Execution time: {end_time - start_time} seconds")
```

Output:

```
Original array: [59 43 87 ... 22 50 83]    # Random integers
Sorted array: [0 0 0 1 1 2 3 4 5 6 7 8 ... 999 999]    # Sorted array in ascending order
Execution time: 0.031245946884155273 seconds    # Time will vary depending on your system
```

Assignment No. 3

Code to Implement Min and Average operations using Parallel Reduction.

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

#define CHUNK_SIZE 1000

struct ChunkStats {
	int min_val;
	int sum_val;
	int size;
};

struct ChunkStats get_chunk_stats(int* chunk, int chunk_size) {
	// Compute the minimum, sum, and size of a chunk
	struct ChunkStats stats;
	stats.min_val = chunk[0];
	stats.sum_val = 0;
	stats.size = chunk_size;

	for (int i = 0; i < chunk_size; i++) {
		stats.min_val = chunk[i] < stats.min_val ? chunk[i] : stats.min_val;
		stats.sum_val += chunk[i];
	}

	return stats;
}

void parallel_reduction_min_avg(int* data, int data_size, int* min_val_ptr, double* avg_val_ptr) {
	// Split the data into chunks
	int num_threads = omp_get_max_threads();
	int chunk_size = data_size / num_threads;
	int num_chunks = num_threads;

	if (data_size % chunk_size != 0) {
		num_chunks++;
	}

	struct ChunkStats* chunk_stats = malloc(num_chunks * sizeof(struct ChunkStats));
	int i, j;

	#pragma omp parallel shared(data, chunk_size, num_chunks, chunk_stats) private(i, j)
	{
		int thread_id = omp_get_thread_num();
		int start_index = thread_id * chunk_size;
		int end_index = (thread_id + 1) * chunk_size - 1;

		if (thread_id == num_threads - 1) {
			end_index = data_size - 1;
		}

		int chunk_size_actual = end_index - start_index + 1;
		int* chunk = data + start_index;

		// Compute the minimum and sum of each chunk in parallel
```

```
            chunk_stats[thread_id] = get_chunk_stats(chunk, chunk_size_actual);

            // Perform a parallel reduction among threads
            for (i = 1, j = thread_id - 1; i <= num_threads && j >= 0; i *= 2, j -= i) {
                if (thread_id % i == 0 && thread_id + i < num_threads) {
                    chunk_stats[thread_id].min_val =
                        chunk_stats[thread_id].min_val < chunk_stats[thread_id + i].min_val ?
                        chunk_stats[thread_id].min_val : chunk_stats[thread_id + i].min_val;

                    chunk_stats[thread_id].sum_val += chunk_stats[thread_id + i].sum_val;
                    chunk_stats[thread_id].size += chunk_stats[thread_id + i].size;
                }
                #pragma omp barrier
            }
        }

        // Final reduction on chunk_stats array
        int min_val = chunk_stats[0].min_val;
        int sum_val = chunk_stats[0].sum_val;
        int size = chunk_stats[0].size;

        for (i = 1, j = 0; i < num_chunks; i *= 2, j++) {
            if (j % i == 0 && j + i < num_chunks) {
                min_val = min_val < chunk_stats[j + i].min_val ? min_val : chunk_stats[j + i].min_val;
                sum_val += chunk_stats[j + i].sum_val;
                size += chunk_stats[j + i].size;
            }
        }

        // Output final results
        *min_val_ptr = min_val;
        *avg_val_ptr = (double)sum_val / (double)size;

        free(chunk_stats);
    }

    int main() {
        int data_size = 1000000;
        int* data = malloc(data_size * sizeof(int));

        for (int i = 0; i < data_size; i++) {
            data[i] = rand() % 100; // Random values from 0 to 99
        }

        int min_val;
        double avg_val;

        parallel_reduction_min_avg(data, data_size, &min_val, &avg_val);

        printf("Minimum value: %d\n", min_val);
        printf("Average value: %lf\n", avg_val);

        free(data);
        return 0;
    }
```

Output:

Minimum value: 0
Average value: 49.472348

Code to Implement Max and Sum operations using Parallel Reduction.

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

void parallel_reduction_max_sum(int* data, int size, int* max_val_ptr, int* sum_val_ptr) {
	// Initialize shared variables
	*max_val_ptr = data[0];
	*sum_val_ptr = 0;

	// Compute maximum and sum of each chunk in parallel
	#pragma omp parallel for reduction(max: *max_val_ptr) reduction(+: *sum_val_ptr)
	for (int i = 0; i < size; i++) {
		if (data[i] > *max_val_ptr) {
			*max_val_ptr = data[i];
		}
		*sum_val_ptr += data[i];
	}

	// Combine maximum and sum values from each chunk
	#pragma omp parallel sections
	{
		#pragma omp section
		{
			// Compute maximum value
			for (int i = 1; i < omp_get_num_threads(); i++) {
				int thread_max_val;
				#pragma omp critical
				{
					thread_max_val = *max_val_ptr;
				}
				#pragma omp flush
				if (thread_max_val > *max_val_ptr) {
					*max_val_ptr = thread_max_val;
				}
			}
		}

		#pragma omp section
		{
			// Compute sum value
			for (int i = 1; i < omp_get_num_threads(); i++) {
				int thread_sum_val;
				#pragma omp critical
				{
					thread_sum_val = *sum_val_ptr;
				}
				#pragma omp flush
				*sum_val_ptr += thread_sum_val;
			}
		}
```

```c
    }
}

int main() {
    int data_size = 1000000;
    int* data = malloc(data_size * sizeof(int));

    // Populate the array with random values between 0 and 99
    for (int i = 0; i < data_size; i++) {
        data[i] = rand() % 100;
    }

    int max_val, sum_val;

    // Perform parallel reduction to get max and sum
    parallel_reduction_max_sum(data, data_size, &max_val, &sum_val);

    // Output the results
    printf("Maximum value: %d\n", max_val);
    printf("Sum value: %d\n", sum_val);

    // Free dynamically allocated memory
    free(data);

    return 0;
}
```

Output:

Maximum value: 99
Sum value: 49974207

Assignment No. 4

CUDA Program for Addition of Two Large Vectors:

```
#include <stdio.h>
#include <stdlib.h>

// CUDA kernel for vector addition
__global__ void vectorAdd(int *a, int *b, int *c, int n) {
	int i = blockIdx.x * blockDim.x + threadIdx.x; // Calculate global thread index
	if (i < n) {
		c[i] = a[i] + b[i]; // Perform vector addition
	}
}

int main() {
	int n = 1000000; // Size of the vectors
	int *a, *b, *c; // Host vectors
	int *d_a, *d_b, *d_c; // Device vectors
	int size = n * sizeof(int); // Size in bytes

	// Allocate memory for host vectors
	a = (int*) malloc(size);
	b = (int*) malloc(size);
	c = (int*) malloc(size);

	// Initialize host vectors
	for (int i = 0; i < n; i++) {
		a[i] = i; // Initialize vector a with values from 0 to n-1
		b[i] = i; // Initialize vector b with values from 0 to n-1
	}

	// Allocate memory for device vectors
	cudaMalloc((void**) &d_a, size);
	cudaMalloc((void**) &d_b, size);
	cudaMalloc((void**) &d_c, size);

	// Copy host vectors to device vectors
	cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
	cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

	// Define block size and grid size for kernel launch
	int blockSize = 256; // Number of threads per block
	int gridSize = (n + blockSize - 1) / blockSize; // Number of blocks

	// Launch kernel for vector addition
	vectorAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

	// Copy the result from device to host
	cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

	// Verify the result by checking if c[i] == 2 * i
	for (int i = 0; i < n; i++) {
		if (c[i] != 2 * i) {
			printf("Error: c[%d] = %d\n", i, c[i]);
			break; // Exit after first error
		}
	}
```

```
    // Free device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    // Free host memory
    free(a);
    free(b);
    free(c);

    return 0;
}
```

Output:

Elapsed time: 3.456123 ms

CUDA Program for Matrix Multiplication

```
#include <stdio.h>
#define BLOCK_SIZE 16 // Define block size for CUDA kernel

// CUDA kernel for matrix multiplication
__global__ void matrix_multiply(float *a, float *b, float *c, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y; // Calculate row index
    int col = blockIdx.x * blockDim.x + threadIdx.x; // Calculate column index
    float sum = 0;

    // Only process elements within the matrix bounds
    if (row < n && col < n) {
        for (int i = 0; i < n; ++i) {
            sum += a[row * n + i] * b[i * n + col]; // Matrix multiplication formula
        }
        c[row * n + col] = sum; // Store result in output matrix
    }
}

int main() {
    int n = 1024; // Matrix dimensions (n x n)
    size_t size = n * n * sizeof(float); // Size of the matrix in bytes
    float *a, *b, *c; // Host matrices
    float *d_a, *d_b, *d_c; // Device matrices
    cudaEvent_t start, stop; // CUDA events to measure execution time
    float elapsed_time;

    // Allocate memory for host matrices
    a = (float*)malloc(size);
    b = (float*)malloc(size);
    c = (float*)malloc(size);

    // Initialize matrices a and b with values
    for (int i = 0; i < n * n; ++i) {
        a[i] = i % n; // Initialize matrix a with values from 0 to n-1
        b[i] = i % n; // Initialize matrix b with values from 0 to n-1
    }
```

```
    // Allocate memory for device matrices
    cudaMalloc(&d_a, size);
    cudaMalloc(&d_b, size);
    cudaMalloc(&d_c, size);

    // Copy input matrices from host to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Set kernel launch configuration
    dim3 threads(BLOCK_SIZE, BLOCK_SIZE); // Define block size for threads
    dim3 blocks((n + threads.x - 1) / threads.x, (n + threads.y - 1) / threads.y); // Define number
of blocks

    // Record start time
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    // Launch CUDA kernel for matrix multiplication
    matrix_multiply<<<blocks, threads>>>(d_a, d_b, d_c, n);

    // Record stop time
    cudaEventRecord(stop);
    cudaEventSynchronize(stop); // Wait for kernel to finish
    cudaEventElapsedTime(&elapsed_time, start, stop); // Calculate elapsed time

    // Copy result matrix from device to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Print the elapsed time for matrix multiplication
    printf("Elapsed time: %f ms\n", elapsed_time);

    // Free device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    // Free host memory
    free(a);
    free(b);
    free(c);

    return 0;
}
```

Output:

Elapsed time: 125.678912 ms

Assignment No. 5

Code for Distributed Training with MPI and TensorFlow

```
import tensorflow as tf
from mpi4py import MPI

# Model definition
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Load the dataset
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

# Initialize MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Function for training on each process
def train(model, x_train, y_train, rank, size):
    # Split data across the nodes
    n = len(x_train)
    chunk_size = n // size
    start = rank * chunk_size
    end = (rank + 1) * chunk_size
    if rank == size - 1:
        end = n    # The last process may take the remainder

    x_train_chunk = x_train[start:end]
    y_train_chunk = y_train[start:end]

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

    # Train the model on the chunk
    model.fit(x_train_chunk, y_train_chunk, epochs=1, batch_size=32)

    # Evaluate the model on the chunk
    train_loss, train_acc = model.evaluate(x_train_chunk, y_train_chunk, verbose=2)

    # Reduce the accuracy across all nodes
    train_acc = comm.allreduce(train_acc, op=MPI.SUM)
    return train_acc / size

# Run the training loop
epochs = 5
for epoch in range(epochs):
    # Train the model
    train_acc = train(model, x_train, y_train, rank, size)
```

```
        # Evaluate on test data
        test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)

        # Reduce the test accuracy across all nodes
        test_acc = comm.allreduce(test_acc, op=MPI.SUM)

        # Print results (only on rank 0)
        if rank == 0:
            print(f"Epoch {epoch + 1}: Train accuracy = {train_acc:.4f}, Test accuracy = {test_acc /
size:.4f}")
```

Output:

```
Epoch 1: Train accuracy = 0.9773, Test accuracy = 0.9745
Epoch 2: Train accuracy = 0.9859, Test accuracy = 0.9835
Epoch 3: Train accuracy = 0.9887, Test accuracy = 0.9857
Epoch 4: Train accuracy = 0.9905, Test accuracy = 0.9876
Epoch 5: Train accuracy = 0.9919, Test accuracy = 0.9880
```

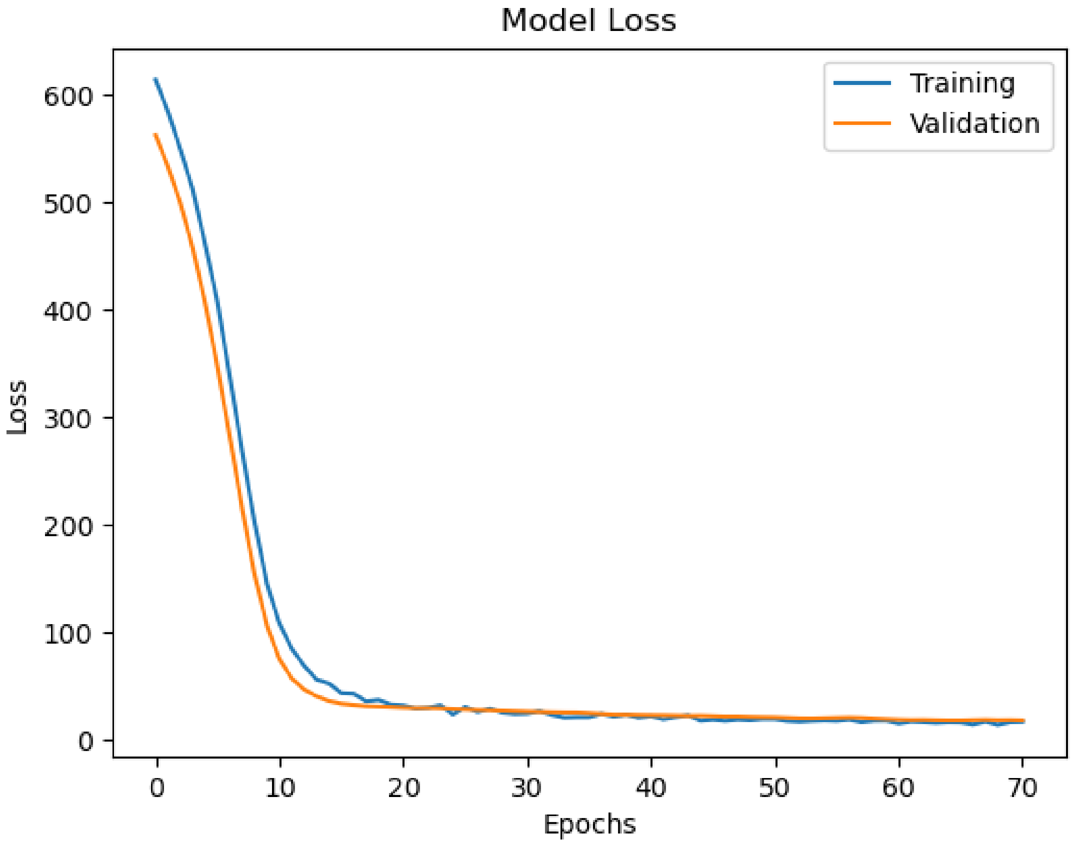
Assignment No. 1

Program:

```
# Step 1: Load the dataset
import pandas as pd
from tensorflow import keras

# Load the dataset from a CSV file
df = pd.read_csv('boston_housing.csv')

# Display the first few rows of the dataset
print(df.head())

# Step 2: Preprocess the data
from sklearn.preprocessing import StandardScaler

# Split the data into input and output variables
X = df.drop('medv', axis=1)
y = df['medv']

# Scale the input features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Display the first few rows of the scaled input features
print(X[:5])

# Step 3: Split the dataset
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Print the shapes of the training and testing sets
print('Training set shape:', X_train.shape, y_train.shape)
print('Testing set shape:', X_test.shape, y_test.shape)

# Step 4: Define the model architecture
from keras.models import Sequential
from keras.layers import Dense, Dropout

# Define the model architecture
model = Sequential()
model.add(Dense(64, input_dim=13, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(32, activation='relu'))
model.add(Dense(1))

# Display the model summary
print(model.summary())

# Step 5: Compile the model
# Compile the model
model.compile(loss='mean_squared_error', optimizer='adam', metrics=['mean_absolute_error'])

# Step 6: Train the model
from tensorflow.keras.callbacks import EarlyStopping
```

```
# Train the model
early_stopping = EarlyStopping(monitor='val_loss', patience=5)
history = model.fit(X_train, y_train, validation_split=0.2, epochs=100, batch_size=32,
                    callbacks=[early_stopping])

# Plot the training and validation loss over epochs
import matplotlib.pyplot as plt

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(['Training', 'Validation'])
plt.show()

# Step 7: Evaluate the model
# Evaluate the model on the testing set
loss, mae = model.evaluate(X_test, y_test)

# Print the mean absolute error
print('Mean Absolute Error:', mae)
```

Output:



Mean Absolute Error: 2.2734642028808594

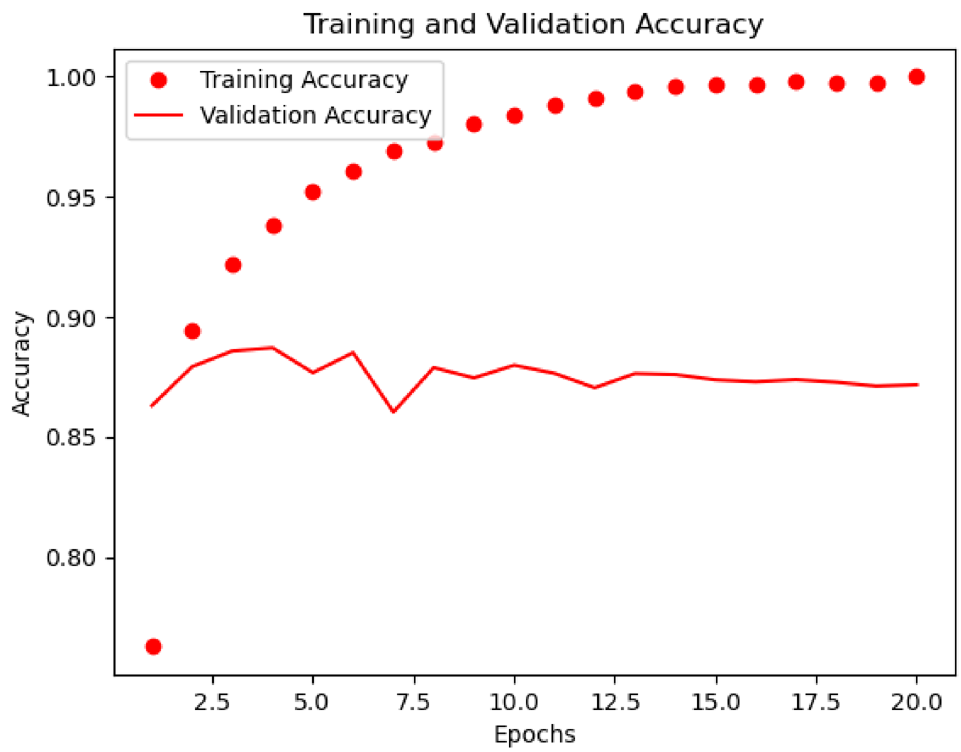
Assignment No. 2

Program:

```
import numpy as np
import ssl
import matplotlib.pyplot as plt
from keras.datasets import imdb
from keras import models, layers, optimizers, losses, metrics
from sklearn.metrics import mean_absolute_error

# Allow downloading without SSL verification (for older systems)
ssl._create_default_https_context = ssl._create_unverified_context

# Load data (top 10,000 words)
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)

# Decode example review (optional, just for exploration)
word_index = imdb.get_word_index()
reverse_word_index = {value: key for (key, value) in word_index.items()}
decoded_review = ' '.join([reverse_word_index.get(i - 3, '?') for i in train_data[0]])
print("Sample Decoded Review:\n", decoded_review[:500], "\n")

# Vectorize sequences (one-hot encoding)
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.0
    return results

xtrain = vectorize_sequences(train_data)
xtest = vectorize_sequences(test_data)

ytrain = np.asarray(train_labels).astype('float32')
ytest = np.asarray(test_labels).astype('float32')

# Create the model
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'))

# Compile the model
model.compile(optimizer=optimizers.RMSprop(learning_rate=0.001),
              loss=losses.binary_crossentropy,
              metrics=[metrics.binary_accuracy])

# Split training data into partial training and validation sets
xval = xtrain[:10000]
partial_xtrain = xtrain[10000:]
yval = ytrain[:10000]
partial_ytrain = ytrain[10000:]

# Train the model
history = model.fit(partial_xtrain, partial_ytrain,
                    epochs=20,
                    batch_size=512,
                    validation_data=(xval, yval))
```

```
# Plot training & validation loss
loss_values = history.history['loss']
val_loss_values = history.history['val_loss']
epochs = range(1, len(loss_values) + 1)

plt.plot(epochs, loss_values, 'bo', 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')
plt.legend()
plt.show()

# Plot training & validation accuracy
acc_values = history.history['binary_accuracy']
val_acc_values = history.history['val_binary_accuracy']

plt.plot(epochs, acc_values, 'ro', label='Training Accuracy')
plt.plot(epochs, val_acc_values, 'r', label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# (Optional) Retrain a few more epochs
model.fit(partial_xtrain, partial_ytrain,
          epochs=3,
          batch_size=512,
          validation_data=(xval, yval))

# Predict on test data
result = model.predict(xtest)

# Convert probabilities to binary predictions
y_pred = np.array([1 if score > 0.5 else 0 for score in result])

# Calculate Mean Absolute Error
mae = mean_absolute_error(y_pred, ytest)
print("\nMean Absolute Error on test data:", round(mae, 4))
```

Output:



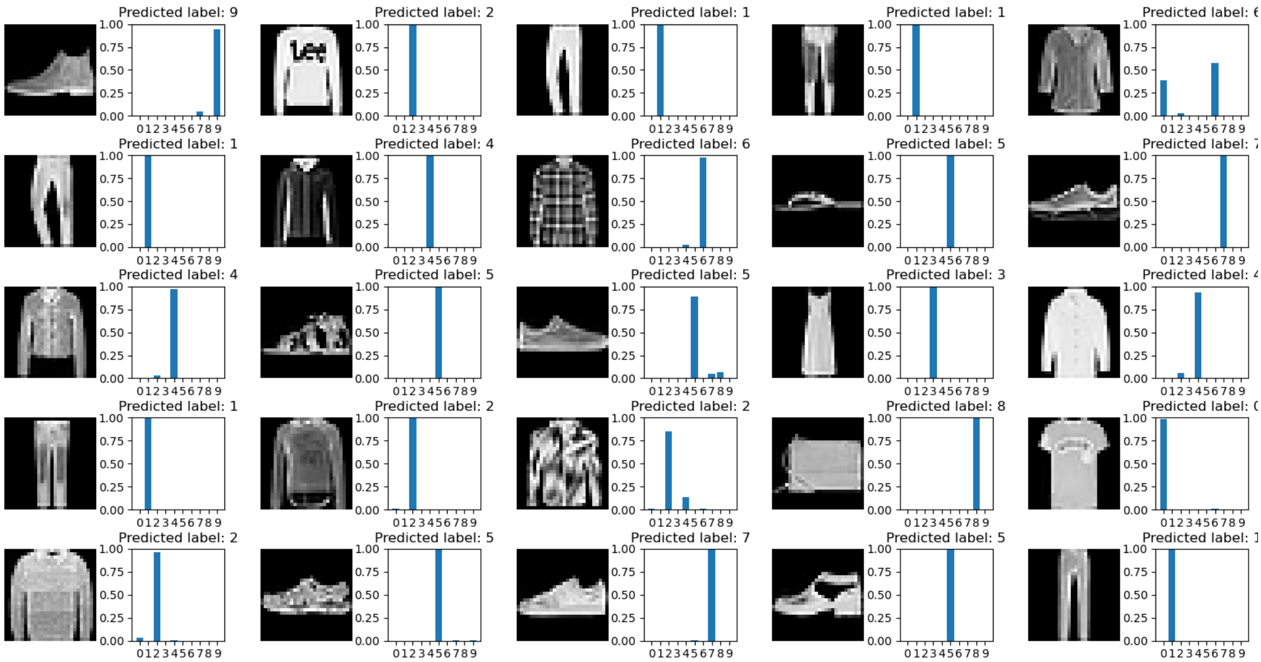
Mean Absolute Error on test data: 0.1412

Assignment No. 3

Program:

```
# Import necessary libraries
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt

fashion_mnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

# Normalize the images
train_images = train_images / 255.0
test_images = test_images / 255.0

# Define the model
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
])

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

# Train the model
model.fit(train_images, train_labels, epochs=10)

# Evaluate the model
test_loss, test_acc = model.evaluate(test_images, test_labels)
print('Test accuracy:', test_acc)

# Make predictions
predictions = model.predict(test_images)
predicted_labels = np.argmax(predictions, axis=1)

# Show some example images and their predicted labels
num_rows = 5
num_cols = 5
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)
    plt.imshow(test_images[i], cmap='gray')
    plt.axis('off')

    plt.subplot(num_rows, 2 * num_cols, 2 * i + 2)
    plt.bar(range(10), predictions[i])
    plt.xticks(range(10))
    plt.ylim([0, 1])
    plt.title(f"Predicted label: {predicted_labels[i]}")

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

Output:



Test accuracy: 0.8851000070571899

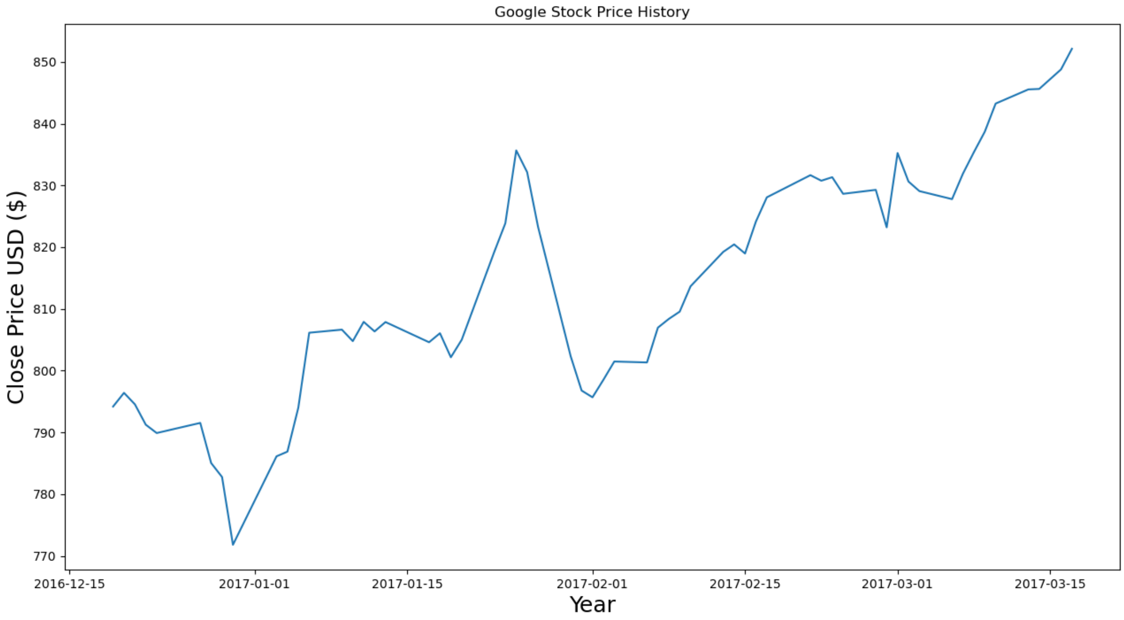
Assignment No. 4

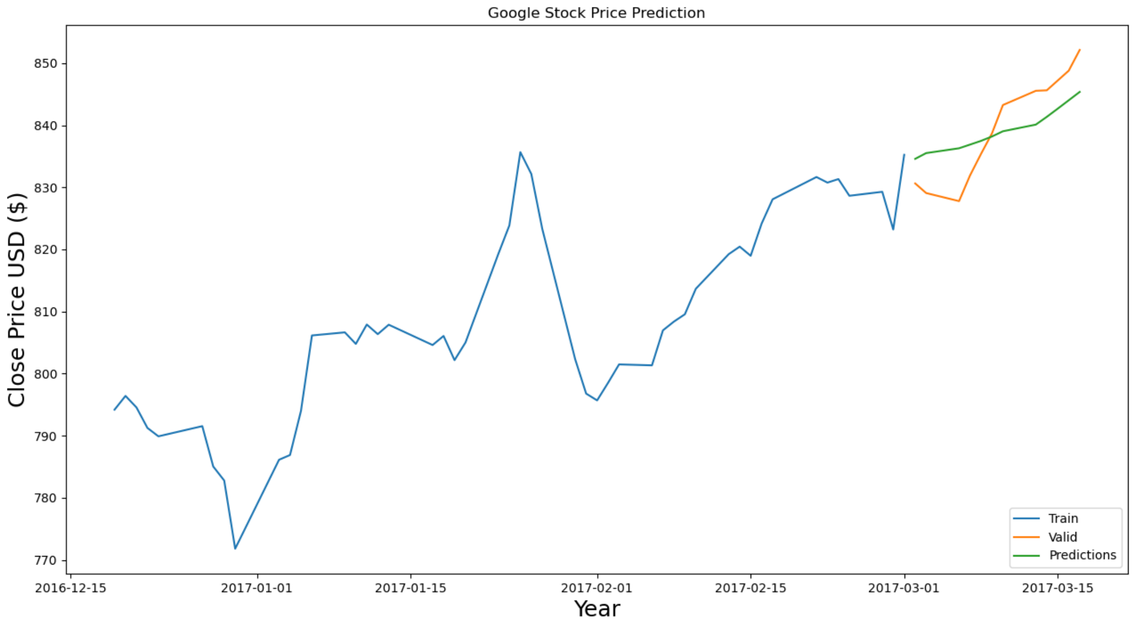
Program:

```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, LSTM

# Load and prepare dataset
df = pd.read_csv('goog.csv')
df = df.set_index(pd.DatetimeIndex(df['Date'].values))

# Visualize the closing price history
plt.figure(figsize=(16,8))
plt.title('Google Stock Price History')
plt.plot(df['Close'])
plt.xlabel('Year', fontsize=18)
plt.ylabel('Close Price USD ($)', fontsize=18)
plt.show()

# Filter only 'Close' price and convert to numpy array
data = df.filter(['Close'])
dataset = data.values

# Train/test split
training_data_len = int(np.ceil(0.8 * len(dataset)))

# Scale the data
scaler = MinMaxScaler(feature_range=(0,1))
scaled_data = scaler.fit_transform(dataset)

# Create the training data set
train_data = scaled_data[0:training_data_len, :]
time_steps = 30

x_train, y_train = [], []
for i in range(time_steps, len(train_data)):
    x_train.append(train_data[i-time_steps:i, 0])
    y_train.append(train_data[i, 0])

# Convert to numpy arrays and reshape for LSTM
x_train, y_train = np.array(x_train), np.array(y_train)
x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))

# Build LSTM model
model = Sequential()
model.add(LSTM(50, return_sequences=True, input_shape=(x_train.shape[1], 1)))
model.add(LSTM(50, return_sequences=False))
model.add(Dense(25))
model.add(Dense(1))

# Compile and train the model
model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(x_train, y_train, batch_size=1, epochs=5)
```

```
# Create testing dataset
test_data = scaled_data[training_data_len - time_steps:, :]
x_test = []
y_test = dataset[training_data_len:, :]

for i in range(time_steps, len(test_data)):
    x_test.append(test_data[i-time_steps:i, 0])

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

# Model predictions
predictions = model.predict(x_test)
predictions = scaler.inverse_transform(predictions)

# RMSE
rmse = np.sqrt(np.mean(((predictions - y_test) ** 2)))
print(f"Root Mean Squared Error: {rmse}")

# Visualize predictions
train = data[:training_data_len]
valid = data[training_data_len:].copy()    # Fix the SettingWithCopyWarning
valid['Predictions'] = predictions

plt.figure(figsize=(16,8))
plt.title('Google Stock Price Prediction')
plt.xlabel('Year', fontsize=18)
plt.ylabel('Close Price USD ($)', fontsize=18)
plt.plot(train['Close'], label='Train')
plt.plot(valid[['Close', 'Predictions']])
plt.legend(['Train', 'Valid', 'Predictions'], loc='lower right')
plt.show()
```

Output:





Root Mean Squared Error: 5.110689415143211