Assignment No. 1

Depth First Search

Code -

```
#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();

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

	s.pop();
	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]) {
		s.push(adj_node);
		}
	}
	}
	}
}
```

```
int main() {
        int n, m, start_node;
        cout<<"Enter no. of Node,no. of Edges and Starting Node of graph:\n";
        cin >> n >> m >> start_node;
       //n: node,m:edges
      cout<<"Enter pair of node and edges:\n";

        for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;

//u and v: Pair of edges
        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 -

C:\Users\TEJALS~1\AppData\Local\Temp\Rar$DIa21352.27419\DFS.cpp - [Executing] - Dev-C++ 5.11
File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 4.9.2 64-bit Release
(globals)
Project
dfs (int node
main () : int
graph [MAX
MAX : int
visited [MAX
graph:\n";
62 }
63
C:\Users\TEJALS~1\AppData\Local\Temp\Rar$DIa21352.27419\DFS.exe
Enter no. of Node,no. of Edges and Starting Node of graph:
4 3 0
Enter pair of node and edges:
0 1
0 2
2 4
0 2 4 1

Process exited after 18.9 seconds with return value 0
Press any key to continue . . .

Breadth First Search

Code -

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


class node
{
   public:

          node *left, *right;
          int data;

};

class Breadthfs
{

 public:

 node *insert(node *, int);
 void bfs(node *);

};


node *insert(node *root, int data)
// inserts a node in tree
{

          if(!root)
          {

                    root=new node;
                    root->left=NULL;
                    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==NULL)
        {

                temp->left=new node;
                temp->left->left=NULL;
                temp->left->right=NULL;
                temp->left->data=data;
                return root;
        }
        else
        {

        q.push(temp->left);

        }

        if(temp->right==NULL)
        {

                temp->right=new node;
                temp->right->left=NULL;
                temp->right->right=NULL;
                temp->right->data=data;
                return root;
        }
        else
        {
```

```
                q.push(temp->right);

                }

        }

}


void bfs(node *head)
{

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

                int qSize;

                while (!q.empty())
                {
                        qSize = q.size();
                        #pragma omp parallel for
            //creates parallel threads
                        for (int i = 0; i < qSize; i++)
                        {
                                node* currNode;
                                #pragma omp critical
                                {
                                  currNode = q.front();
                                  q.pop();
                                  cout<<"\t"<<currNode->data;

                                }// prints parent node
                                #pragma omp critical
                                {
                                if(currNode->left)// push parent's left node in queue
                                        q.push(currNode->left);
                                if(currNode->right)
                                        q.push(currNode->right);
                                }// push parent's right node in queue
```

```
                        }
                }

}

int main(){

        node *root=NULL;
        int data;
        char ans;

        do
        {
                cout<<"\n enter data=>";
                cin>>data;

                root=insert(root,data);

                cout<<"do you want insert one more node?";
                cin>>ans;

        }while(ans=='y'||ans=='Y');

        bfs(root);

        return 0;
}
```

Output -

TDM-GCC 4.9.2 64-bit Release
(globals)
Project
Breadthfs
node : class
bfs (node *he
insert (node *
main () : int
C:\Users\TEJALS~1\AppData\Local\Temp\Rar$DIa21352.10678\BFS.exe
enter data=> 5
do you want insert one more node? y
enter data=> 3
do you want insert one more node? y
enter data=> 2
do you want insert one more node? y
enter data=> 1
do you want insert one more node? y
enter data=> 7
do you want insert one more node? y
enter data=> 8
do you want insert one more node? n
5 3 2 1 7 8

Process exited after 50.64 seconds with return value 0
Press any key to continue . . .
// inserts a node in tree
{

Assignment No. 2

Bubble Sort

Code -

```
import time
import random

#timer to keep track of performance
start = time.perf_counter()

# a function to implement bubble sort in parallel
def Parallel_bubble_sort(lst):
    # variable to keep track of swaps to end the while loop
        Sorted = 0

    # variable to get length of list
        n = len(lst)

    #loop to traverse all list elements in phases
        while Sorted == 0:

        # set to 1 initially to assume list is sorted
        # and no swaps occurred
                Sorted = 1

        # traverse all list elements in pair
        # start at index 0 for odd phase
        # start at index 1 for even phase
                for i in range(0, n-1, 2):
            # check if current element greater than next element
                        if lst[i] > lst[i+1]:
                # if so, swap the elements
                                lst[i], lst[i+1] = lst[i+1], lst[i]

                # set to 0 to imply a swap occurred
                                Sorted = 0
                for i in range(1, n-1, 2):
                        if lst[i] > lst[i+1]:
                                lst[i], lst[i+1] = lst[i+1], lst[i]
                                Sorted = 0
```

```
        # print final sorted list
        print(lst)

# an example list to test above program
lst = [(random.randint(0,100)) for i in range(100)]

Parallel_Bubble_Sort(lst)

finish = time.perf_counter()

print(f'Finished in {round(finish-start,2)} second(s)')
```

Output -

Merge Sort

Code -

```
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m

    # create temp arrays
    L = [0] * (n1)
    R = [0] * (n2)

    # Copy data to temp arrays L[] and R[]
    for i in range(0, n1):
        L[i] = arr[l + i]

    for j in range(0, n2):
        R[j] = arr[m + 1 + j]

    # Merge the temp arrays back into arr[l..r]
    i = 0     # Initial index of first subarray
    j = 0     # Initial index of second subarray
    k = l     # Initial index of merged subarray

    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    # Copy the remaining elements of L[], if there
    # are any
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    # Copy the remaining elements of R[], if there
    # are any
    while j < n2:
        arr[k] = R[j]
```

```
            j += 1
            k += 1

# l is for left index and r is right index of the
# sub-array of arr to be sorted


def mergeSort(arr, l, r):
    if l < r:

            # Same as (l+r)//2, but avoids overflow for
            # large l and h
            m = l+(r-l)//2

            # Sort first and second halves
            mergeSort(arr, l, m)
            mergeSort(arr, m+1, r)
            merge(arr, l, m, r)


# Driver code to test above
arr = [12, 11, 13, 5, 6, 7]
n = len(arr)
print("Given array is")
for i in range(n):
    print("%d" % arr[i],end=" ")

mergeSort(arr, 0, n-1)
print("\n\nSorted array is")
for i in range(n):
    print("%d" % arr[i],end=" ")
```

Output -

File Edit View Navigate Code Refactor Run Tools VCS Window Help
pythonProject - main.py
pythonProject
main.py
Project
pythonProject C:\Users\Vaishnavi Nighot\Pycharm
venv library root
main.py
External Libraries
Scratches and Consoles
def merge(arr, l, m, r):
n1 = m - l + 1
n2 = r - m
L = [0] * (n1)
R = [0] * (n2)
for i in range(0, n1):
L[i] = arr[l + i]
for j in range(0, n2):
R[j] = arr[m + 1 + j]
i = 0
j = 0
k = l
while i < n1 and j < n2:
if L[i] <= R[j]:
merge()
Run: main
"C:\Users\Vaishnavi Nighot\PycharmProjects\pythonProject\venv\Scripts\python.exe" "C:/Users/Vaishnavi Nighot/PycharmProjects/pythonProject/main.py
Given array is
12 11 13 5 6 7
Sorted array is
5 6 7 11 12 13
Process finished with exit code 0
Version Control
Run
TODO
Problems
Terminal
Python Packages
Python Console
Services
Download pre-built shared indexes: Reduce the indexing time and CPU load with pre-built Python packages shared indexes // Always download /... (16 minutes ago)
10:10 CRLF UTF-8 4 spaces Python 3.9 (pythonProject)
75°F
Clear
Search
ENG
IN
10:30 PM
5/1/2023

Assignment No. 3

Min, Max, Sum and Average operations using Parallel Reduction

Code -

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

using namespace std;

void min_reduction(vector<int>& arr) {
  int min_value = INT_MAX;
  #pragma omp parallel for reduction(min: min_value)
  for (int i = 0; i < arr.size(); i++) {
    if (arr[i] < min_value) {
      min_value = arr[i];
    }
  }
  cout << "Minimum value: " << min_value << endl;
}

void max_reduction(vector<int>& arr) {
  int max_value = INT_MIN;
  #pragma omp parallel for reduction(max: max_value)
  for (int i = 0; i < arr.size(); i++) {
    if (arr[i] > max_value) {
      max_value = arr[i];
    }
  }
  cout << "Maximum value: " << max_value << endl;
}

void sum_reduction(vector<int>& arr) {
  int sum = 0;
   #pragma omp parallel for reduction(+: sum)
   for (int i = 0; i < arr.size(); i++) {
    sum += arr[i];
  }
  cout << "Sum: " << sum << endl;
}
```

```
void average_reduction(vector<int>& arr) {
  int sum = 0;
  #pragma omp parallel for reduction(+: sum)
  for (int i = 0; i < arr.size(); i++) {
    sum += arr[i];
  }
  cout << "Average: " << (double)sum / arr.size() << endl;
}

int main() {
  vector<int> arr;
  arr.push_back(5);
  arr.push_back(2);
  arr.push_back(9);
  arr.push_back(1);
  arr.push_back(7);
  arr.push_back(6);
  arr.push_back(8);
  arr.push_back(3);
  arr.push_back(4);

  min_reduction(arr);
  max_reduction(arr);
  sum_reduction(arr);
  average_reduction(arr);
}
```

Output -

File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 4.9.2 64-bit Release
(globals)
Project
average_rec
main () : int
max_reduct
min_reducti
sum_reduct
C:\Users\TEJALS~1\AppData\Local\Temp\Rar$DIa21352.10678\BFS.exe
Minimum value: 1
Maximum value: 9
Sum: 45
Average: 5

Process exited after 0.1638 seconds with return value 0
Press any key to continue . . .
56 arr.push_back(8);
57 arr.push_back(3);
58 arr.push_back(4);

```
!nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Wed_Sep_21_10:33:58_PDT_2022
Cuda compilation tools, release 11.8, V11.8.89
Build cuda_11.8.r11.8/compiler.31833905_0
```

```
!pip install git+https://github.com/andreinechaev/nvcc4jupyter.git
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting git+https://github.com/andreinechaev/nvcc4jupyter.git
  Cloning https://github.com/andreinechaev/nvcc4jupyter.git to /tmp/pip-req-build-6qcw7cp3
  Running command git clone --filter=blob:none --quiet https://github.com/andreinechaev/nvcc4jupyter.git /tmp/pip-req-build-6qcw7cp
  Resolved https://github.com/andreinechaev/nvcc4jupyter.git to commit aac710a35f52bb78ab34d2e52517237941399eff
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: NVCCPlugin
  Building wheel for NVCCPlugin (setup.py) ... done
  Created wheel for NVCCPlugin: filename=NVCCPlugin-0.0.2-py3-none-any.whl size=4305 sha256=dd25b7ca6620d9871d8e0415e4abeb1aa2505e2
  Stored in directory: /tmp/pip-ephem-wheel-cache-fwapyafc/wheels/a8/b9/18/23f8ef71ceb0f63297dd1903aedd067e6243a68ea756d6feea
Successfully built NVCCPlugin
Installing collected packages: NVCCPlugin
Successfully installed NVCCPlugin-0.0.2
```

```
%load_ext nvcc_plugin
```

```
created output directory at /content/src
Out bin /content/result.out
```

```
%%cu
#include <cuda_runtime.h>
#include <iostream>

__global__ void matmul(int* A, int* B, int* C, int N) {
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    int Col = blockIdx.x*blockDim.x+threadIdx.x;
    if (Row < N && Col < N) {
        int Pvalue = 0;
        for (int k = 0; k < N; k++) {
            Pvalue += A[Row*N+k] * B[k*N+Col];
        }
        C[Row*N+Col] = Pvalue;
    }
}

int main() {
    int N = 512;
    int size = N * N * sizeof(int);
    int* A, * B, * C;
    int* dev_A, * dev_B, * dev_C;
    cudaMallocHost(&A, size);
    cudaMallocHost(&B, size);
    cudaMallocHost(&C, size);
    cudaMalloc(&dev_A, size);
    cudaMalloc(&dev_B, size);
    cudaMalloc(&dev_C, size);

    // Initialize matrices A and B
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i*N+j] = i*N+j;
            B[i*N+j] = j*N+i;
        }
    }

    cudaMemcpy(dev_A, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_B, B, size, cudaMemcpyHostToDevice);

    dim3 dimBlock(16, 16);
    dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);

    matmul<<<dimGrid, dimBlock>>>(dev_A, dev_B, dev_C, N);

    cudaMemcpy(C, dev_C, size, cudaMemcpyDeviceToHost);

    // Print the result
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            std::cout << C[i*N+j] << " ";
```

```
        }
        std::cout << std::endl;
    }

    // Free memory
    cudaFree(dev_A);
    cudaFree(dev_B);
    cudaFree(dev_C);
    cudaFreeHost(A);
    cudaFreeHost(B);
    cudaFreeHost(C);

    return 0;
}
```

```
 44608256 111586048 178563840 245541632 312519424 379497216 446475008 513452800 580430592 647408384
 111586048 312781568 513977088 715172608 916368128 1117563648 1318759168 1519954688 1721150208 1922345728
 178563840 513977088 849390336 1184803584 1520216832 1855630080 -2103923968 -1768510720 -1433097472 -1097684224
 245541632 715172608 1184803584 1654434560 2124065536 -1701270784 -1231639808 -762008832 -292377856 177253120
 312519424 916368128 1520216832 2124065536 -1567053056 -963204352 -359355648 244493056 848341760 1452190464
 379497216 1117563648 1855630080 -1701270784 -963204352 -225137920 512928512 1250994944 1989061376 -1567839488
 446475008 1318759168 -2103923968 -1231639808 -359355648 512928512 1385212672 -2037470464 -1165186304 -292902144
 513452800 1519954688 -1768510720 -762008832 244493056 1250994944 -2037470464 -1030968576 -24466688 982035200
 580430592 1721150208 -1433097472 -292377856 848341760 1989061376 -1165186304 -24466688 1116252928 -2037994752
 647408384 1922345728 -1097684224 177253120 1452190464 -1567839488 -292902144 982035200 -2037994752 -763057408
```

```
%%cu
#include <iostream>
#include <cuda_runtime.h>

using namespace std;

__global__ void addVectors(int* A, int* B, int* C, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
    {
        C[i] = A[i] + B[i];
    }
}

int main()
{
    int n = 1000000;
    int* A, * B, * C;
    int size = n * sizeof(int);

    // Allocate memory on the host
    cudaMallocHost(&A, size);
    cudaMallocHost(&B, size);
    cudaMallocHost(&C, size);

    // Initialize the vectors
    for (int i = 0; i < n; i++)
    {
        A[i] = i;
        B[i] = i * 2;
    }
    // Allocate memory on the device
    int* dev_A, * dev_B, * dev_C;
    cudaMalloc(&dev_A, size);
    cudaMalloc(&dev_B, size);
    cudaMalloc(&dev_C, size);

    // Copy data from host to device
    cudaMemcpy(dev_A, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_B, B, size, cudaMemcpyHostToDevice);

    // Launch the kernel
    int blockSize = 256;
    int numBlocks = (n + blockSize - 1) / blockSize;
    addVectors<<<numBlocks, blockSize>>>(dev_A, dev_B, dev_C, n);

    // Copy data from device to host
    cudaMemcpy(C, dev_C, size, cudaMemcpyDeviceToHost);

    // Print the results
    for (int i = 0; i < 10; i++)
    {
        cout << C[i] << " ";
    }
    cout << endl;
```

```
    // Free memory
    cudaFree(dev_A);
    cudaFree(dev_B);
    cudaFree(dev_C);
    cudaFreeHost(A);
    cudaFreeHost(B);
    cudaFreeHost(C);

    return 0;
}
```

```
 0 3 6 9 12 15 18 21 24 27
```

Colab paid products - Cancel contracts here
✓ 0s completed at 14:27

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt
from sklearn import metrics

# Load the OCR dataset

# The MNIST dataset is a built-in dataset provided by Keras.
# It consists of 70,000 28x28 grayscale images, each of which displays a single handwritten digit from 0 to 9.
# The training set consists of 60,000 images, while the test set has 10,000 images.

(x_train, y_train), (x_test, y_test) = mnist.load_data()

# X_train and X_test are our array of images while y_train and y_test are our array of labels for each image.
# The first tuple contains the training set features (X_train) and the training set labels (y_train).
# The second tuple contains the testing set features (X_test) and the testing set labels (y_test).
# For example, if the image shows a handwritten 7, then the label will be the intger 7.

plt.imshow(x_train[0], cmap='gray')  # imshow() function which simply displays an image.
plt.show() # cmap is responsible for mapping a specific colormap to the values found in the array that you passed as the first argument.

# image appears black and white and that each axis of the plot ranges from 0 to 28.

# This is because of the format that all the images in the dataset have:

# 1. All the images are grayscale, meaning they only contain black, white and grey.
# 2. The images are 28 pixels by 25 pixels in size (28x28).

print(x_train[0])

# image data is just an array of digits. You can almost make out a 5 from the pattern of the digits in the array.
# Array of 28 values
# a grayscale pixel is stored as a digit between 0 and 255 where 0 is black, 255 is white and values in between are different shades of g
# Therefore, each value in the [28][28] array tells the computer which color to put in that position when we display the actual image.
```

```
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   3  18  18  18 126 136
  175  26 166 255 247 127   0   0   0   0]
```

```
# reformat our X_train array and our X_test array because they do not have the correct shape.
# Reshape the data to fit the model
print("X_train shape", x_train.shape)
print("y_train shape", y_train.shape)
print("X_test shape", x_test.shape)
print("y_test shape", y_test.shape)


# Here you can see that for the training sets we have 60,000 elements and the testing sets have 10,000 elements.

# y_train and y_test only have 1 dimensional shapes because they are just the labels of each element.

# x_train and x_test have 3 dimensional shapes because they have a width and height (28x28 pixels) for each element.

# (60000, 28, 28) 1st parameter in the tuple shows us how much image we have 2nd and 3rd parameters are the pixel values from x to y (28x
# The pixel value varies between 0 to 255.
# (60000,) Training labels with integers from 0-9 with dtype of uint8. It has the shape (60000,).
# (10000, 28, 28) Testing data that consists of grayscale images. It has the shape (10000, 28, 28) and the dtype of uint8. The pixel valu
# (10000,) Testing labels that consist of integers from 0-9 with dtype uint8. It has the shape (10000,).
```

```
X_train shape (60000, 28, 28)
y_train shape (60000,)
X_test shape (10000, 28, 28)
y_test shape (10000,)
```

```
  250 182   0   0   0   0   0   0   0   0]
```

```
# X: Training data of shape (n_samples, n_features)
# y: Training label values of shape (n_samples, n_labels)
# 2D array of height and width, 28 pixels by 28 pixels will just become 784 pixels (28 squared).
# Remember that X_train has 60,000 elemenets, each with 784 total pixels so will become shape (60000, 784).
# Whereas X_test has 10,000 elements, each with each with 784 total pixels so will become shape (10000, 784).

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32') # use 32-bit precision when training a neural network, so at one point the training data will have to
x_test = x_test.astype('float32')
x_train /= 255  # Each image has Intensity from 0 to 255
x_test /= 255

# Regarding the division by 255, this is the maximum value of a byte (the input feature's type before the conversion to float32),
# so this will ensure that the input features are scaled between 0.0 and 1.0.
# USING svm-https://mgta.gmu.edu/courses/ml-with-python/handwrittenDigitRecognition.php#:~:text=Remember%20that%20X_train%20has%2060%2C00


# Convert class vectors to binary class matrices
num_classes = 10
y_train = np.eye(num_classes)[y_train]  # Return a 2-D array with ones on the diagonal and zeros elsewhere.
y_test = np.eye(num_classes)[y_test] # f your particular categories is present then it mark as 1 else 0 in remain row
```

```
# Define the model architecture
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))  # The input_shape argument is passed to the foremost layer. It comprises of
model.add(Dropout(0.2)) # DROP OUT RATIO 20%
model.add(Dense(512, activation='relu')) #returns a sequence of vectors of dimension 512
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation='softmax'))


# Compile the model
model.compile(loss='categorical_crossentropy',  # for a multi-class classification problem
              optimizer=RMSprop(),
              metrics=['accuracy'])


# Train the model
batch_size = 128 # batch_size argument is passed to the layer to define a batch size for the inputs.
epochs = 20
history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1, # verbose=1 will show you an animated progress bar eg. [==========]
                    validation_data=(x_test, y_test)) # Using validation_data means you are providing the training set and validation set
                                                      # validation_split means you only provide a training set and keras splits it into a
```

```
Epoch 1/20
469/469 [==============================] - 12s 24ms/step - loss: 0.2539 - accuracy: 0.9216 - val_loss: 0.1443 - val_accuracy: 0.954
Epoch 2/20
469/469 [==============================] - 9s 20ms/step - loss: 0.1039 - accuracy: 0.9665 - val_loss: 0.0730 - val_accuracy: 0.9789
Epoch 3/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0742 - accuracy: 0.9767 - val_loss: 0.0693 - val_accuracy: 0.979
Epoch 4/20
469/469 [==============================] - 11s 22ms/step - loss: 0.0565 - accuracy: 0.9827 - val_loss: 0.0669 - val_accuracy: 0.981
Epoch 5/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0480 - accuracy: 0.9850 - val_loss: 0.0747 - val_accuracy: 0.979
Epoch 6/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0396 - accuracy: 0.9872 - val_loss: 0.0699 - val_accuracy: 0.981
Epoch 7/20
469/469 [==============================] - 10s 20ms/step - loss: 0.0342 - accuracy: 0.9889 - val_loss: 0.0687 - val_accuracy: 0.983
Epoch 8/20
469/469 [==============================] - 11s 22ms/step - loss: 0.0294 - accuracy: 0.9903 - val_loss: 0.0661 - val_accuracy: 0.983
Epoch 9/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0255 - accuracy: 0.9923 - val_loss: 0.0745 - val_accuracy: 0.982
Epoch 10/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0220 - accuracy: 0.9930 - val_loss: 0.0734 - val_accuracy: 0.983
Epoch 11/20
469/469 [==============================] - 9s 19ms/step - loss: 0.0202 - accuracy: 0.9934 - val_loss: 0.0679 - val_accuracy: 0.9843
Epoch 12/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0179 - accuracy: 0.9944 - val_loss: 0.0721 - val_accuracy: 0.985
Epoch 13/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0170 - accuracy: 0.9946 - val_loss: 0.0684 - val_accuracy: 0.985
Epoch 14/20
469/469 [==============================] - 11s 23ms/step - loss: 0.0147 - accuracy: 0.9949 - val_loss: 0.0841 - val_accuracy: 0.982
Epoch 15/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0138 - accuracy: 0.9956 - val_loss: 0.0802 - val_accuracy: 0.983
Epoch 16/20
469/469 [==============================] - 10s 21ms/step - loss: 0.0139 - accuracy: 0.9956 - val_loss: 0.0712 - val_accuracy: 0.984
Epoch 17/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0124 - accuracy: 0.9959 - val_loss: 0.0695 - val_accuracy: 0.985
Epoch 18/20
469/469 [==============================] - 10s 22ms/step - loss: 0.0120 - accuracy: 0.9960 - val_loss: 0.0772 - val_accuracy: 0.986
Epoch 19/20
469/469 [==============================] - 11s 23ms/step - loss: 0.0094 - accuracy: 0.9968 - val_loss: 0.0829 - val_accuracy: 0.985
Epoch 20/20
469/469 [==============================] - 9s 20ms/step - loss: 0.0095 - accuracy: 0.9969 - val_loss: 0.0791 - val_accuracy: 0.9843
```

```
# Evaluate the model
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 0.07907029986381531
Test accuracy: 0.9843000173568726
```

```
# The IMDB sentiment classification dataset consists of 50,000 movie reviews from IMDB users that are labeled as either positive (1) or r
# The reviews are preprocessed and each one is encoded as a sequence of word indexes in the form of integers.
# The words within the reviews are indexed by their overall frequency within the dataset. For example, the integer “2” encodes the seconc
# The 50,000 reviews are split into 25,000 for training and 25,000 for testing.
# Text Process word by word at diffrent timestamp ( You may use RNN(Recurrent Neural Network) LSTM(Long Short-Term Memory) GRU(Gated Recu
# convert input text to vector reprent input text
# DOMAIN: Digital content and entertainment industry
# CONTEXT: The objective of this project is to build a text classification model that analyses the customer's sentiments based on their r
# DATA DESCRIPTION: The Dataset of 50,000 movie reviews from IMDB, labelled by sentiment (positive/negative).
# Reviews have been preprocessed, and each review is encoded as a sequence of word indexes (integers).
# For convenience, the words are indexed by their frequency in the dataset, meaning the for that has index 1 is the most frequent word.
# Use the first 20 words from each review to speed up training, using a max vocabulary size of 10,000.
# As a convention, "0" does not stand for a specific word, but instead is used to encode any unknown word.

# PROJECT OBJECTIVE: Build a sequential NLP classifier which can use input text parameters to determine the customer sentiments.
import numpy as np
import pandas as pd


#loading imdb data with most frequent 10000 words

from keras.datasets import imdb
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=10000) # you may take top 10,000 word frequently review of movies
#consolidating data for EDA(exploratory data analysis: involves gathering all the relevant data into one place and preparing it for analy
data = np.concatenate((X_train, X_test), axis=0)
label = np.concatenate((y_train, y_test), axis=0)



print("Review is ",X_train[5]) # series of no converted word to vocabulory associated with index
print("Review is ",y_train[5]) # 0 indicating a negative review and 1 indicating a positive review.
```

```
Review is  [1, 778, 128, 74, 12, 630, 163, 15, 4, 1766, 7982, 1051, 2, 32, 85, 156, 45, 40, 148, 139, 121, 664, 665, 10, 10, 1361,
Review is  0
```

```
vocab=imdb.get_word_index() #The code you provided retrieves the word index for the IMDB dataset
print(vocab)
```

```
{'fawn': 34701, 'tsukino': 52006, 'nunnery': 52007, 'sonja': 16816, 'vani': 63951, 'woods': 1408, 'spiders': 16115, 'hanging': 2345
```

```
data #data is a numpy array that contains all the text data from the IMDB dataset, both the training and testing sets.
```

```
array([list([1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838, 112, 50, 670,
2, 9, 35, 480, 284, 5, 150, 4, 172, 112, 167, 2, 336, 385, 39, 4, 172, 4536, 1111, 17, 546, 38, 13, 447, 4, 192, 50, 16, 6, 147,
2025, 19, 14, 22, 4, 1920, 4613, 469, 4, 22, 71, 87, 12, 16, 43, 530, 38, 76, 15, 13, 1247, 4, 22, 17, 515, 17, 12, 16, 626, 18,
2, 5, 62, 386, 12, 8, 316, 8, 106, 5, 4, 2223, 5244, 16, 480, 66, 3785, 33, 4, 130, 12, 16, 38, 619, 5, 25, 124, 51, 36, 135, 48,
25, 1415, 33, 6, 22, 12, 215, 28, 77, 52, 5, 14, 407, 16, 82, 2, 8, 4, 107, 117, 5952, 15, 256, 4, 2, 7, 3766, 5, 723, 36, 71, 43,
530, 476, 26, 400, 317, 46, 7, 4, 2, 1029, 13, 104, 88, 4, 381, 15, 297, 98, 32, 2071, 56, 26, 141, 6, 194, 7486, 18, 4, 226, 22,
21, 134, 476, 26, 480, 5, 144, 30, 5535, 18, 51, 36, 28, 224, 92, 25, 104, 4, 226, 65, 16, 38, 1334, 88, 12, 16, 283, 5, 16, 4472,
113, 103, 32, 15, 16, 5345, 19, 178, 32]),
       list([1, 194, 1153, 194, 8255, 78, 228, 5, 6, 1463, 4369, 5012, 134, 26, 4, 715, 8, 118, 1634, 14, 394, 20, 13, 119, 954,
189, 102, 5, 207, 110, 3103, 21, 14, 69, 188, 8, 30, 23, 7, 4, 249, 126, 93, 4, 114, 9, 2300, 1523, 5, 647, 4, 116, 9, 35, 8163,
4, 229, 9, 340, 1322, 4, 118, 9, 4, 130, 4901, 19, 4, 1002, 5, 89, 29, 952, 46, 37, 4, 455, 9, 45, 43, 38, 1543, 1905, 398, 4,
1649, 26, 6853, 5, 163, 11, 3215, 2, 4, 1153, 9, 194, 775, 7, 8255, 2, 349, 2637, 148, 605, 2, 8003, 15, 123, 125, 68, 2, 6853,
15, 349, 165, 4362, 98, 5, 4, 228, 9, 43, 2, 1157, 15, 299, 120, 5, 120, 174, 11, 220, 175, 136, 50, 9, 4373, 228, 8255, 5, 2,
656, 245, 2350, 5, 4, 9837, 131, 152, 491, 18, 2, 32, 7464, 1212, 14, 9, 6, 371, 78, 22, 625, 64, 1382, 9, 8, 168, 145, 23, 4,
1690, 15, 16, 4, 1355, 5, 28, 6, 52, 154, 462, 33, 89, 78, 285, 16, 145, 95]),
       list([1, 14, 47, 8, 30, 31, 7, 4, 249, 108, 7, 4, 5974, 54, 61, 369, 13, 71, 149, 14, 22, 112, 4, 2401, 311, 12, 16, 3711,
33, 75, 43, 1829, 296, 4, 86, 320, 35, 534, 19, 263, 4821, 1301, 4, 1873, 33, 89, 78, 12, 66, 16, 4, 360, 7, 4, 58, 316, 334, 11,
4, 1716, 43, 645, 662, 8, 257, 85, 1200, 42, 1228, 2578, 83, 68, 3912, 15, 36, 165, 1539, 278, 36, 69, 2, 780, 8, 106, 14, 6905,
1338, 18, 6, 22, 12, 215, 28, 610, 40, 6, 87, 326, 23, 2300, 21, 23, 22, 12, 272, 40, 57, 31, 11, 4, 22, 47, 6, 2307, 51, 9, 170,
23, 595, 116, 595, 1352, 13, 191, 79, 638, 89, 2, 14, 9, 8, 106, 607, 624, 35, 534, 6, 227, 7, 129, 113]),
       ...,
       list([1, 13, 1408, 15, 8, 135, 14, 9, 35, 32, 46, 394, 20, 62, 30, 5093, 21, 45, 184, 78, 4, 1492, 910, 769, 2290, 2515,
395, 4257, 5, 1454, 11, 119, 2, 89, 1036, 4, 116, 218, 78, 21, 407, 100, 30, 128, 262, 15, 7, 185, 2280, 284, 1842, 2, 37, 315, 4,
226, 20, 272, 2942, 40, 29, 152, 60, 181, 8, 30, 50, 553, 362, 80, 119, 12, 21, 846, 5518]),
       list([1, 11, 119, 241, 9, 4, 840, 20, 12, 468, 15, 94, 3684, 562, 791, 39, 4, 86, 107, 8, 97, 14, 31, 33, 4, 2960, 7, 743,
46, 1028, 9, 3531, 5, 4, 768, 47, 8, 79, 90, 145, 164, 162, 50, 6, 501, 119, 7, 9, 4, 78, 232, 15, 16, 224, 11, 4, 333, 20, 4,
985, 200, 5, 2, 5, 9, 1861, 8, 79, 357, 4, 20, 47, 220, 57, 206, 139, 11, 12, 5, 55, 117, 212, 13, 1276, 92, 124, 51, 45, 1188,
71, 536, 13, 520, 14, 20, 6, 2302, 7, 470]),
       list([1, 6, 52, 7465, 430, 22, 9, 220, 2594, 8, 28, 2, 519, 3227, 6, 769, 15, 47, 6, 3482, 4067, 8, 114, 5, 33, 222, 31,
55, 184, 704, 5586, 2, 19, 346, 3153, 5, 6, 364, 350, 4, 184, 5586, 9, 133, 1810, 11, 5417, 2, 21, 4, 7298, 2, 570, 50, 2005,
2643, 9, 6, 1249, 17, 6, 2, 2, 21, 17, 6, 1211, 232, 1138, 2249, 29, 266, 56, 96, 346, 194, 308, 9, 194, 21, 29, 218, 1078, 19, 4,
78, 173, 7, 27, 2, 5698, 3406, 718, 2, 9, 6, 6907, 17, 210, 5, 3281, 5677, 47, 77, 395, 14, 172, 173, 18, 2740, 2931, 4517, 82,
127, 27, 173, 11, 6, 392, 217, 21, 50, 9, 57, 65, 12, 2, 53, 40, 35, 390, 7, 11, 4, 3567, 7, 4, 314, 74, 6, 792, 22, 2, 19, 714,
727, 5205, 382, 4, 91, 6533, 439, 19, 14, 20, 9, 1441, 5805, 1118, 4, 756, 25, 124, 4, 31, 12, 16, 93, 804, 34, 2005, 2643])],
      dtype=object)
```

```
label #label is a numpy array that contains all the sentiment labels from the IMDB dataset, both the training and testing sets.0 indicate
```

```
array([1, 0, 0, ..., 0, 0, 0])
```

```
X_train.shape
```

```
(25000,)
```

```
X_test.shape
```

```
(25000,)
```

```
y_train
```

```
array([1, 0, 0, ..., 0, 1, 0])
```

```
y_test # y_test is 25000, which indicates that it contains 25000 sentiment labels, one for each review in the testing set.
```

```
array([0, 1, 1, ..., 0, 0, 0])
```

```
# Function to perform relevant sequence adding on the data
# Now it is time to prepare our data. We will vectorize every review and fill it with zeros so that it contains exactly 1000 numbers.
# That means we fill every review that is shorter than 500 with zeros.
# We do this because the biggest review is nearly that long and every input for our neural network needs to have the same size.
# We also transform the targets into floats.
# sequences is name of method the review less than 1000 we perform padding overthere

def vectorize(sequences, dimension = 10000):
  # Create an all-zero matrix of shape (len(sequences), dimension)
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1
    return results

# The script transforms your dataset into a binary vector space model.
# First, if we examine the x_train content we see that each review is represented as a sequence of word ids. Each word id corresponds to
# print(train_data[0])  # print the first review
# [1, 14, 22, 16, 43, 530, 973, ..., 5345, 19, 178, 32]
# the size of the entire dictionary, dictionary=10000 in your example.
# We will then associate each element/index of this vector with one word/word_id.
# So word represented by word id 14 will now be represented by 14-th element of this vector.


# Each element will either be 0 (word is not present in the review) or 1 (word is present in the review).
# And we can treat this as a probability, so we even have meaning for values in between 0 and 1.
# Furthermore, every review will now be represented by this very long (sparse) vector which has a constant length for every review.
# word      word_id
# I      -> 0
# you    -> 1
# he     -> 2
# be     -> 3
# eat    -> 4
# happy  -> 5
# sad    -> 6
# banana -> 7
# a      -> 8

# the sentences would then be processed in a following way.


# I be happy      -> [0,3,5]   -> [1,0,0,1,0,1,0,0,0]
# I eat a banana. -> [0,4,8,7] -> [1,0,0,0,1,0,0,1,1]

# Now I highlighted the word sparse.
# That means, there will have A LOT MORE zeros in comparison with ones.
# We can take advantage of that. Instead of checking every word, whether it is contained in a review or not;
# we will check a substantially smaller list of only those words that DO appear in our review.

# Therefore, we can make things easy for us and create reviews × vocabulary matrix of zeros right away by np.zeros((len(sequences), dimer
# And then just go through words in each review and flip the indicator to 1.0 at position corresponding to that word:

# result[review_id][word_id] = 1.0
# So instead of doing 25000 x 10000 = 250 000 000 operations,
# we only did number of words = 5 967 841. That's just ~2.5% of original amount of operations.


# The for loop here is not processing all the matrix. As you can see, it enumerates elements of the sequence, so it's looping only on one
# t = np.array([1,2,3,4,5,6,7,8,9])
# r = np.zeros((len(t), 10))

#Output
```

```
# array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
#   [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
#   [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
#   [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
#  [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
#  [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
#  [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
#  [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
# [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]) #

# then we modify elements with the same way you have :

# for i, s in enumerate(t):
#  r[i,s] = 1.
# array([[0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
#   [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
#   [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
#   [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
#   [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
#   [0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
#   [0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
#   [0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
#   [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])
  # you can see that the for loop modified only a set of elements (len(t))
  # which has index [i,s] (in this case ; (0, 1), (1, 2), (2, 3), an so on))


# Now we split our data into a training and a testing set.
# The training set will contain  reviews and the testing set

test_x = data[:10000]
test_y = label[:10000]
train_x = data[10000:]
train_y = label[10000:]
```

```
test_x
```

```
array([list([1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838, 112, 50, 670,
2, 9, 35, 480, 284, 5, 150, 4, 172, 112, 167, 2, 336, 385, 39, 4, 172, 4536, 1111, 17, 546, 38, 13, 447, 4, 192, 50, 16, 6, 147,
2025, 19, 14, 22, 4, 1920, 4613, 469, 4, 22, 71, 87, 12, 16, 43, 530, 38, 76, 15, 13, 1247, 4, 22, 17, 515, 17, 12, 16, 626, 18,
2, 5, 62, 386, 12, 8, 316, 8, 106, 5, 4, 2223, 5244, 16, 480, 66, 3785, 33, 4, 130, 12, 16, 38, 619, 5, 25, 124, 51, 36, 135, 48,
25, 1415, 33, 6, 22, 12, 215, 28, 77, 52, 5, 14, 407, 16, 82, 2, 8, 4, 107, 117, 5952, 15, 256, 4, 2, 7, 3766, 5, 723, 36, 71, 43,
530, 476, 26, 400, 317, 46, 7, 4, 2, 1029, 13, 104, 88, 4, 381, 15, 297, 98, 32, 2071, 56, 26, 141, 6, 194, 7486, 18, 4, 226, 22,
21, 134, 476, 26, 480, 5, 144, 30, 5535, 18, 51, 36, 28, 224, 92, 25, 104, 4, 226, 65, 16, 38, 1334, 88, 12, 16, 283, 5, 16, 4472,
113, 103, 32, 15, 16, 5345, 19, 178, 32]),
       list([1, 194, 1153, 194, 8255, 78, 228, 5, 6, 1463, 4369, 5012, 134, 26, 4, 715, 8, 118, 1634, 14, 394, 20, 13, 119, 954,
189, 102, 5, 207, 110, 3103, 21, 14, 69, 188, 8, 30, 23, 7, 4, 249, 126, 93, 4, 114, 9, 2300, 1523, 5, 647, 4, 116, 9, 35, 8163,
4, 229, 9, 340, 1322, 4, 118, 9, 4, 130, 4901, 19, 4, 1002, 5, 89, 29, 952, 46, 37, 4, 455, 9, 45, 43, 38, 1543, 1905, 398, 4,
1649, 26, 6853, 5, 163, 11, 3215, 2, 4, 1153, 9, 194, 775, 7, 8255, 2, 349, 2637, 148, 605, 2, 8003, 15, 123, 125, 68, 2, 6853,
15, 349, 165, 4362, 98, 5, 4, 228, 9, 43, 2, 1157, 15, 299, 120, 5, 120, 174, 11, 220, 175, 136, 50, 9, 4373, 228, 8255, 5, 2,
656, 245, 2350, 5, 4, 9837, 131, 152, 491, 18, 2, 32, 7464, 1212, 14, 9, 6, 371, 78, 22, 625, 64, 1382, 9, 8, 168, 145, 23, 4,
1690, 15, 16, 4, 1355, 5, 28, 6, 52, 154, 462, 33, 89, 78, 285, 16, 145, 95]),
       list([1, 14, 47, 8, 30, 31, 7, 4, 249, 108, 7, 4, 5974, 54, 61, 369, 13, 71, 149, 14, 22, 112, 4, 2401, 311, 12, 16, 3711,
33, 75, 43, 1829, 296, 4, 86, 320, 35, 534, 19, 263, 4821, 1301, 4, 1873, 33, 89, 78, 12, 66, 16, 4, 360, 7, 4, 58, 316, 334, 11,
4, 1716, 43, 645, 662, 8, 257, 85, 1200, 42, 1228, 2578, 83, 68, 3912, 15, 36, 165, 1539, 278, 36, 69, 2, 780, 8, 106, 14, 6905,
1338, 18, 6, 22, 12, 215, 28, 610, 40, 6, 87, 326, 23, 2300, 21, 23, 22, 12, 272, 40, 57, 31, 11, 4, 22, 47, 6, 2307, 51, 9, 170,
23, 595, 116, 595, 1352, 13, 191, 79, 638, 89, 2, 14, 9, 8, 106, 607, 624, 35, 534, 6, 227, 7, 129, 113]),
       ...,
       list([1, 14, 9, 6, 66, 327, 5, 1047, 20, 15, 4, 436, 223, 70, 358, 45, 44, 107, 2515, 5, 6, 1132, 37, 26, 623, 245, 8, 412,
19, 294, 334, 18, 6, 117, 137, 21, 4, 1389, 92, 391, 5, 36, 1090, 5, 140, 8, 169, 4, 223, 23, 68, 205, 4, 1132, 9, 773, 5621, 5,
59, 456, 56, 8, 41, 403, 580, 9, 4, 1155, 912, 37, 694, 6, 176, 44, 113, 23, 4, 1004, 7, 4, 6567, 2694, 9, 4, 922, 5, 2, 912, 37,
5190, 183, 276, 148, 289, 295, 23, 35, 1154, 5, 12, 166, 18, 6, 654, 5, 253, 1061, 58, 50, 26, 57, 318, 302, 7, 4, 6849, 728, 38,
12, 218, 954, 33, 32, 45, 4, 118, 662, 1626, 20, 15, 207, 110, 38, 230, 45, 6, 66, 52, 20, 18, 2166]),
       list([1, 14, 20, 9, 43, 160, 856, 206, 509, 21, 12, 100, 28, 77, 38, 76, 128, 54, 4, 1865, 216, 46, 36, 66, 887, 49, 3822,
339, 294, 40, 1798, 2, 37, 93, 15, 530, 206, 720, 11, 3567, 17, 36, 847, 56, 4, 890, 39, 4, 4565, 62, 28, 679, 4, 753, 206, 844,
83, 142, 318, 88, 4, 360, 7, 4, 22, 16, 2659, 727, 21, 1753, 128, 74, 25, 62, 535, 39, 14, 552, 7, 20, 95, 385, 4, 477, 136, 4,
123, 180, 4, 31, 75, 69, 77, 1064, 18, 21, 16, 40, 149, 142, 39, 4, 6, 768, 11, 4, 2084, 36, 1258, 5261, 164, 1936, 5, 36, 521,
187, 6, 313, 269, 8, 516, 257, 85, 172, 154, 172, 154]),
       list([1, 51, 527, 487, 5, 116, 57, 1613, 51, 25, 191, 97, 6, 52, 20, 19, 6, 686, 109, 7660, 12, 16, 224, 11, 2, 19, 532,
807, 10, 10, 38, 14, 554, 271, 23, 6, 1189, 8, 67, 27, 336, 4, 554, 1655, 304, 6, 1707, 5, 4, 1811, 47, 6, 483, 1274, 5, 1442,
1696, 2817, 38, 4, 554, 6141, 11, 6, 2144, 5, 6095, 95, 29, 1129, 187, 4247, 11, 4, 5152, 366, 29, 214, 6590, 10, 10, 315, 15, 58,
29, 1860, 6, 2123, 1453, 4, 86, 58, 29, 1860, 12, 125, 11, 4, 2144, 4, 333, 58, 29, 166, 6, 2, 46, 7, 6, 9459, 5, 9866, 4, 2123,
107, 665, 7, 1214, 541, 2, 46, 7, 4, 2, 4539, 1578, 72, 7, 6498, 2, 4, 3942, 9997, 10, 10, 82, 4, 554, 1068, 8, 1968, 6, 2, 19,
727, 1901, 10, 10, 3288])],
      dtype=object)
```

```
test_y
```

```
array([1, 0, 0, ..., 1, 0, 0])
```

```
train_x
```

```
array([list([1, 13, 104, 14, 9, 31, 7, 4, 4343, 7, 4, 3776, 3394, 2, 495, 103, 141, 87, 2048, 17, 76, 2, 44, 164, 525, 13, 197,
14, 16, 338, 4, 177, 16, 6118, 5253, 2, 2, 2, 21, 61, 1126, 2, 16, 15, 36, 4621, 19, 4, 2, 157, 5, 605, 46, 49, 7, 4, 297, 8, 276,
```

```
       11, 4, 621, 837, 844, 10, 10, 25, 43, 92, 81, 2282, 5, 95, 947, 19, 4, 297, 806, 21, 15, 9, 43, 355, 13, 119, 49, 3636, 6951, 43,
       40, 4, 375, 415, 21, 2, 92, 947, 19, 4, 2282, 1771, 14, 5, 106, 2, 1151, 48, 25, 181, 8, 67, 6, 530, 9089, 1253, 7, 4, 2]),
              list([1, 14, 20, 16, 835, 835, 835, 51, 6, 1703, 56, 51, 6, 387, 180, 32, 812, 57, 2327, 6, 394, 437, 7, 676, 5, 58, 62,
       24, 386, 12, 8, 61, 5301, 912, 37, 80, 106, 233]),
              list([1, 86, 125, 13, 62, 40, 8, 213, 46, 15, 137, 13, 244, 24, 35, 2809, 4, 96, 4, 3100, 16, 2400, 80, 2384, 129, 1663,
       4633, 4, 2, 115, 2085, 15, 2, 2, 165, 495, 9123, 18, 199, 4, 2, 88, 36, 70, 79, 35, 1271, 5, 4, 4824, 18, 24, 116, 23, 14, 17,
       160, 2, 301, 2, 2799, 16, 2085, 9508, 4129, 36, 343, 3973, 17, 4, 2, 37, 47, 965, 602, 5, 60, 80, 2, 11, 4, 3100, 63, 9, 43, 379,
       48, 4, 2619, 69, 1668, 90, 8, 2, 15, 96, 36, 62, 28, 839, 11, 294, 9954, 900, 36, 62, 30, 43, 2254, 18, 35, 1271, 2, 14, 506, 16,
       115, 1803, 3383, 1204, 44, 4, 2, 34, 4, 2, 2, 1373, 7, 4, 1494, 525, 5, 60, 54, 1803, 34, 4, 4824, 3383, 1204, 40, 54, 12, 645,
       29, 100, 24, 1527, 48, 15, 218, 3793, 824, 13, 92, 124, 51, 9, 5, 6, 3275, 2408, 62, 28, 1840, 35, 2, 10, 10, 1324, 347, 12, 517,
       125, 73, 19, 4, 2, 7, 112, 4, 4069, 7, 6, 506, 19, 2, 21, 4, 3100, 166, 14, 20, 5028, 1297]),
              ...,
              list([1, 13, 1408, 15, 8, 135, 14, 9, 35, 32, 46, 394, 20, 62, 30, 5093, 21, 45, 184, 78, 4, 1492, 910, 769, 2290, 2515,
       395, 4257, 5, 1454, 11, 119, 2, 89, 1036, 4, 116, 218, 78, 21, 407, 100, 30, 128, 262, 15, 7, 185, 2280, 284, 1842, 2, 37, 315, 4,
       226, 20, 272, 2942, 40, 29, 152, 60, 181, 8, 30, 50, 553, 362, 80, 119, 12, 21, 846, 5518]),
              list([1, 11, 119, 241, 9, 4, 840, 20, 12, 468, 15, 94, 3684, 562, 791, 39, 4, 86, 107, 8, 97, 14, 31, 33, 4, 2960, 7, 743,
       46, 1028, 9, 3531, 5, 4, 768, 47, 8, 79, 90, 145, 164, 162, 50, 6, 501, 119, 7, 9, 4, 78, 232, 15, 16, 224, 11, 4, 333, 20, 4,
       985, 200, 5, 2, 5, 9, 1861, 8, 79, 357, 4, 20, 47, 220, 57, 206, 139, 11, 12, 5, 55, 117, 212, 13, 1276, 92, 124, 51, 45, 1188,
       71, 536, 13, 520, 14, 20, 6, 2302, 7, 470]),
              list([1, 6, 52, 7465, 430, 22, 9, 220, 2594, 8, 28, 2, 519, 3227, 6, 769, 15, 47, 6, 3482, 4067, 8, 114, 5, 33, 222, 31,
       55, 184, 704, 5586, 2, 19, 346, 3153, 5, 6, 364, 350, 4, 184, 5586, 9, 133, 1810, 11, 5417, 2, 21, 4, 7298, 2, 570, 50, 2005,
       2643, 9, 6, 1249, 17, 6, 2, 2, 21, 17, 6, 1211, 232, 1138, 2249, 29, 266, 56, 96, 346, 194, 308, 9, 194, 21, 29, 218, 1078, 19, 4,
       78, 173, 7, 27, 2, 5698, 3406, 718, 2, 9, 6, 6907, 17, 210, 5, 3281, 5677, 47, 77, 395, 14, 172, 173, 18, 2740, 2931, 4517, 82,
       127, 27, 173, 11, 6, 392, 217, 21, 50, 9, 57, 65, 12, 2, 53, 40, 35, 390, 7, 11, 4, 3567, 7, 4, 314, 74, 6, 792, 22, 2, 19, 714,
       727, 5205, 382, 4, 91, 6533, 439, 19, 14, 20, 9, 1441, 5805, 1118, 4, 756, 25, 124, 4, 31, 12, 16, 93, 804, 34, 2005, 2643])],
      dtype=object)
```

```
train_y
```

```
array([0, 0, 0, ..., 0, 0, 0])
```

```
print("Categories:", np.unique(label))
print("Number of unique words:", len(np.unique(np.hstack(data))))
# The hstack() function is used to stack arrays in sequence horizontally (column wise).

#>>> import numpy as np
#>>> x = np.array((3,5,7))
#>>> y = np.array((5,7,9))
#>>> np.hstack((x,y))
# array([3, 5, 7, 5, 7, 9])


# You can see in the output above that the dataset is labeled into two categories, either 0 or 1,
# which represents the sentiment of the review.

# The whole dataset contains 9998 unique words and the average review length is 234 words, with a standard deviation of 173 words.
```

```
Categories: [0 1]
Number of unique words: 9998
```

```
length = [len(i) for i in data]
print("Average Review length:", np.mean(length))
print("Standard Deviation:", round(np.std(length)))

# The whole dataset contains 9998 unique words and the average review length is 234 words, with a standard deviation of 173 words.
```

```
Average Review length: 234.75892
Standard Deviation: 173
```

```
# If you look at the data you will realize it has been already pre-processed.
# All words have been mapped to integers and the integers represent the words sorted by their frequency.
# This is very common in text analysis to represent a dataset like this.
# So 4 represents the 4th most used word,
# 5 the 5th most used word and so on...
# The integer 1 is reserved for the start marker,
# the integer 2 for an unknown word and 0 for padding.

# Let's look at a single training example:

print("Label:", label[0])
```

```
Label: 1
```

```
print(data[0])
```

```
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838, 112, 50, 670, 2, 9, 35, 48
```

```
# Let's decode the first review
# Above you see the first review of the dataset which is labeled as positive (1).
```

```
# The code below retrieves the dictionary mapping word indices back into the original words so that we can read them.
# It replaces every unknown word with a “#”. It does this by using the get_word_index() function.


# Retrieves a dict mapping words to their index in the IMDB dataset.
index = imdb.get_word_index()
# If there is a possibility of multiple keys with the same value, you will need to specify the desired behaviour in this case to lookup m
# ivd = dict((v, k) for k, v in d.items())
# If you want to peek at the reviews yourself and see what people have actually written, you can reverse the process too:
reverse_index = dict([(value, key) for (key, value) in index.items()])
decoded = " ".join( [reverse_index.get(i - 3, "#") for i in data[0]] ) #The purpose of subtracting 3 from i is to adjust the indice,# to
print(decoded)
```

```
# this film was just brilliant casting location scenery story direction everyone's really suited the part they played and you could
```

```
index
```

```
{'fawn': 34701,
 'tsukino': 52006,
 'nunnery': 52007,
 'sonja': 16816,
 'vani': 63951,
 'woods': 1408,
 'spiders': 16115,
 'hanging': 2345,
 'woody': 2289,
 'trawling': 52008,
 "hold's": 52009,
 'comically': 11307,
 'localized': 40830,
 'disobeying': 30568,
 "'royale": 52010,
 "harpo's": 40831,
 'canet': 52011,
 'aileen': 19313,
 'acurately': 52012,
 "diplomat's": 52013,
 'rickman': 25242,
 'arranged': 6746,
 'rumbustious': 52014,
 'familiarness': 52015,
 "spider'": 52016,
 'hahahah': 68804,
 "wood'": 52017,
 'transvestism': 40833,
 "hangin'": 34702,
 'bringing': 2338,
 'seamier': 40834,
 'wooded': 34703,
 'bravora': 52018,
 'grueling': 16817,
 'wooden': 1636,
 'wednesday': 16818,
 "'prix": 52019,
 'altagracia': 34704,
 'circuitry': 52020,
 'crotch': 11585,
 'busybody': 57766,
 "tart'n'tangy": 52021,
 'burgade': 14129,
 'thrace': 52023,
 "tom's": 11038,
 'snuggles': 52025,
 'francesco': 29114,
 'complainers': 52027,
 'templarios': 52125,
 '272': 40835,
 '273': 52028,
 'zaniacs': 52130,
 '275': 34706,
 'consenting': 27631,
 'snuggled': 40836,
 'inanimate': 15492,
 'uality': 52030,
 'bronte': 11926,
```

```
reverse_index
```

```
{34701: 'fawn',
 52006: 'tsukino',
 52007: 'nunnery',
 16816: 'sonja',
 63951: 'vani',
 1408: 'woods',
 16115: 'spiders',
 2345: 'hanging',
```

```
 2289: 'woody',
 52008: 'trawling',
 52009: "hold's",
 11307: 'comically',
 40830: 'localized',
 30568: 'disobeying',
 52010: "'royale",
 40831: "harpo's",
 52011: 'canet',
 19313: 'aileen',
 52012: 'acurately',
 52013: "diplomat's",
 25242: 'rickman',
 6746: 'arranged',
 52014: 'rumbustious',
 52015: 'familiarness',
 52016: "spider'",
 68804: 'hahahah',
 52017: "wood'",
 40833: 'transvestism',
 34702: "hangin'",
 2338: 'bringing',
 40834: 'seamier',
 34703: 'wooded',
 52018: 'bravora',
 16817: 'grueling',
 1636: 'wooden',
 16818: 'wednesday',
 52019: "'prix",
 34704: 'altagracia',
 52020: 'circuitry',
 11585: 'crotch',
 57766: 'busybody',
 52021: "tart'n'tangy",
 14129: 'burgade',
 52023: 'thrace',
 11038: "tom's",
 52025: 'snuggles',
 29114: 'francesco',
 52027: 'complainers',
 52125: 'templarios',
 40835: '272',
 52028: '273',
 52130: 'zaniacs',
 34706: '275',
 27631: 'consenting',
 40836: 'snuggled',
 15492: 'inanimate',
 52030: 'uality',
 11926: 'bronte',
```

```
decoded
```

```
'# this film was just brilliant casting location scenery story direction everyone's really suited the part they played and you cou
ld just imagine being there robert # is an amazing actor and now the same being director # father came from the same scottish isla
nd as myself so i loved the fact there was a real connection with this film the witty remarks throughout the film were great it wa
s just brilliant so much that i bought the film as soon as it was released for # and would recommend it to everyone to watch and t
he fly fishing was amazing really cried at the end it was so sad and you know what they say if you cry at a film it must have been
good and this definitely was also # to the two little boy's that played the # of norman and paul they were just brilliant children
are often left out of the # list i think because the stars that play them all grown up are such a big profile for the whole film b
```

```
#Adding sequence to data
# Vectorization is the process of converting textual data into numerical vectors and is a process that is usually applied once the text i
data = vectorize(data)
label = np.array(label).astype("float32")

# Now it is time to prepare our data. We will vectorize every review and fill it with zeros so that it contains exactly 1000 numbers.
# That means we fill every review that is shorter than 1000 with zeros.
# We do this because the biggest review is nearly that long and every input for our neural network needs to have the same size.
# We also transform the targets into floats.
```

```
data
```

```
array([[0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.],
       ...,
       [0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.]])
```

```
label
```

```
array([1., 0., 0., ..., 0., 0., 0.], dtype=float32)
```

```
# Let's check distribution of data

# To create plots for EDA(exploratory data analysis)
import seaborn as sns #seaborn is a popular Python visualization library that is built on top of Matplotlib and provides a high-level int
sns.set(color_codes=True)
import matplotlib.pyplot as plt # %matplotlib to display Matplotlib plots inline with the notebook
%matplotlib inline


labelDF=pd.DataFrame({'label':label})
sns.countplot(x='label', data=labelDF)

# For below analysis it is clear that data has equel distribution of sentiments.This will help us building a good model.
```

```
<Axes: xlabel='label', ylabel='count'>
```

```
# Creating train and test data set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(data,label, test_size=0.20, random_state=1)


X_train.shape
```

```
(40000, 10000)
```

```
X_test.shape
```

```
(10000, 10000)
```

```
# Let's create sequential model,In deep learning, a Sequential model is a linear stack of layers, where you can simply add layers one aft

from keras.utils import to_categorical
from keras import models
from keras import layers


model = models.Sequential()
# Input - Layer
# Note that we set the input-shape to 10,000 at the input-layer because our reviews are 10,000 integers long.
# The input-layer takes 10,000 as input and outputs it with a shape of 50.
model.add(layers.Dense(50, activation = "relu", input_shape=(10000, )))
# Hidden - Layers
# Please note you should always use a dropout rate between 20% and 50%. # here in our case 0.3 means 30% dropout we are using dropout to
# By the way, if you want you can build a sentiment analysis without LSTMs(Long Short-Term Memory networks), then you simply need to repl
model.add(layers.Dropout(0.3, noise_shape=None, seed=None))
model.add(layers.Dense(50, activation = "relu")) #ReLU" stands for Rectified Linear Unit, and it is a commonly used activation function i
model.add(layers.Dropout(0.2, noise_shape=None, seed=None))
model.add(layers.Dense(50, activation = "relu"))
# Output- Layer
model.add(layers.Dense(1, activation = "sigmoid")) #adds another Dense layer to the model, but with a single neuron instead of 50,i.e. ou
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 50)                500050

 dropout (Dropout)           (None, 50)                0

 dense_1 (Dense)             (None, 50)                2550

 dropout_1 (Dropout)         (None, 50)                0

 dense_2 (Dense)             (None, 50)                2550

 dense_3 (Dense)             (None, 1)                 51

=================================================================
```

```
Total params: 505,201
Trainable params: 505,201
Non-trainable params: 0
_________________________________________________________________
```

```
#For early stopping
# Stop training when a monitored metric has stopped improving.
# monitor: Quantity to be monitored.
# patience: Number of epochs with no improvement after which training will be stopped.
import tensorflow as tf #TensorFlow provides a wide range of tools and features for data processing, model building, model training, and
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3)
```

```
# We use the “adam” optimizer, an algorithm that changes the weights and biases during training.
# During training, the weights and biases of a machine learning model are updated iteratively to minimize the difference between the mode
# We also choose binary-crossentropy as loss (because we deal with binary classification) and accuracy as our evaluation metric.

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

```
# Now we're able to train our model. We'll do this with a batch_size of 500 and only for two epochs because I recognized that the model c
# batch size defines the number of samples that will be propagated through the network.
# For instance, let's say you have 1050 training samples and you want to set up a batch_size equal to 100.
# The algorithm takes the first 100 samples (from 1st to 100th) from the training dataset and trains the network.
# Next, it takes the second 100 samples (from 101st to 200th) and trains the network again.
# We can keep doing this procedure until we have propagated all samples through of the network.
# Problem might happen with the last set of samples. In our example, we've used 1050 which is not divisible by 100 without remainder.
# The simplest solution is just to get the final 50 samples and train the network.
##The goal is to find the number of epochs that results in good performance on a validation dataset without overfitting to the training c
results = model.fit(
 X_train, y_train,
 epochs= 2,
 batch_size = 500,
 validation_data = (X_test, y_test),
 callbacks=[callback]
)
```

```
Epoch 1/2
80/80 [==============================] - 8s 80ms/step - loss: 0.4230 - accuracy: 0.8036 - val_loss: 0.2635 - val_accuracy: 0.8920
Epoch 2/2
80/80 [==============================] - 4s 55ms/step - loss: 0.2198 - accuracy: 0.9169 - val_loss: 0.2569 - val_accuracy: 0.8987
```

```
# Let's check mean accuracy of our model
print(np.mean(results.history["val_accuracy"])) # Good model should have a mean accuracy significantly higher than 50%
```

```
0.8953500092029572
```

```
#Let's plot training history of our model

# list all data in history
print(results.history.keys())
# summarize history for accuracy
plt.plot(results.history['accuracy']) #Plots the training accuracy of the model at each epoch.
plt.plot(results.history['val_accuracy']) #Plots the validation accuracy of the model at each epoch.
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(results.history['loss'])
plt.plot(results.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

```
model.predict(X_test)
```

```
313/313 [==============================] - 2s 6ms/step
array([[0.17430142],
       [0.9865479 ],
       [0.86662334],
       ...,
       [0.9772741 ],
       [0.9754161 ],
       [0.9927141 ]], dtype=float32)
```

```
#Out put analysis,
#[0.9865479] is a single prediction value for a particular input sample in the test data.
#It is the predicted probability of the positive sentiment class (class 1) for that input.
#Since the output activation function of the last layer of the model is sigmoid, which maps the predicted values to a range of [0,1]
#,the output values represent the probabilities of the positive class.
#In this case, the probability of the positive class for the given input sample is 0.9865479, which is very close to 1, indicating that t
```

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

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

# There are 10 image classes in this dataset and each class has a mapping corresponding to the following labels:

#0 T-shirt/top
#1 Trouser
#2 pullover
#3 Dress
#4 Coat
#5 sandals
#6 shirt
#7 sneaker
#8 bag
#9 ankle boot

# https://ml-course.github.io/master/09%20-%20Convolutional%20Neural%20Networks.pdf
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 1us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 [==============================] - 2s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 [==============================] - 1s 0us/step
```

```
plt.imshow(x_train[1])
```

```
<matplotlib.image.AxesImage at 0x7f555d19d730>
```

```
plt.imshow(x_train[0])
```

```
<matplotlib.image.AxesImage at 0x7f5557360b50>
```

```
# Next, we will preprocess the data by scaling the pixel values to be between 0 and 1, and then reshaping the images to be 28x28 pixels.

x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

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

# 28, 28 comes from width, height, 1 comes from the number of channels
# -1 means that the length in that dimension is inferred.
# This is done based on the constraint that the number of elements in an ndarray or Tensor when reshaped must remain the same.

# each image is a row vector (784 elements) and there are lots of such rows (let it be n, so there are 784n elements). So TensorFlow can
```

```
# converting the training_images array to 4 dimensional array with sizes 60000, 28, 28, 1 for 0th to 3rd dimension.

x_train.shape
```

```
(60000, 28, 28, 1)
```

```
x_test.shape
```

```
(10000, 28, 28, 1)
```

```
y_train.shape
```

```
(60000,)
```

```
y_test.shape
```

```
(10000,)
```

```
# We will use a convolutional neural network (CNN) to classify the fashion items.
# The CNN will consist of multiple convolutional layers followed by max pooling,
# dropout, and dense layers. Here is the code for the model:

model = keras.Sequential([
    keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
    # 32 filters (default), randomly initialized
    # 3*3 is Size of Filter
    # 28,28,1 size of Input Image
    # No zero-padding: every output 2 pixels less in every dimension
    # in Paramter shwon 320 is value of weights: (3x3 filter weights + 32 bias) * 32 filters
    # 32*3*3=288(Total)+32(bias)= 320


    keras.layers.MaxPooling2D((2,2)),
    # It shown 13 * 13 size image with 32 channel or filter or depth.

    keras.layers.Dropout(0.25),
    # Reduce Overfitting of Training sample drop out 25% Neuron

    keras.layers.Conv2D(64, (3,3), activation='relu'),
    # Deeper layers use 64 filters
    # 3*3 is Size of Filter
    # Observe how the input image on 28x28x1 is transformed to a 3x3x64 feature map
    # 13(Size)-3(Filter Size )+1(bias)=11 Size for Width and Height with 64 Depth or filtter or channel
    # in Paramter shwon 18496 is value of weights: (3x3 filter weights + 64 bias) * 64 filters
    # 64*3*3=576+1=577*32 + 32(bias)=18496

    keras.layers.MaxPooling2D((2,2)),
    # It shown 5 * 5 size image with 64 channel or filter or depth.

    keras.layers.Dropout(0.25),

    keras.layers.Conv2D(128, (3,3), activation='relu'),
    # Deeper layers use 128 filters
    # 3*3 is Size of Filter
    # Observe how the input image on 28x28x1 is transformed to a 3x3x128 feature map
    # It show 5(Size)-3(Filter Size )+1(bias)=3 Size for Width and Height with 64 Depth or filtter or channel
    # 128*3*3=1152+1=1153*64 + 64(bias)= 73856

    # To classify the images, we still need a Dense and Softmax layer.
    # We need to flatten the 3x3x128 feature map to a vector of size 1152
    # https://medium.com/@iamvarman/how-to-calculate-the-number-of-parameters-in-the-cnn-5bd55364d7ca
```

```
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu'),
    # 128 Size of Node in Dense Layer
    # 1152*128 = 147584

    keras.layers.Dropout(0.25),
    keras.layers.Dense(10, activation='softmax')
    # 10 Size of Node another Dense Layer
    # 128*10+10 bias= 1290
])
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 dropout (Dropout)           (None, 13, 13, 32)        0

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 dropout_1 (Dropout)         (None, 5, 5, 64)          0

 conv2d_2 (Conv2D)           (None, 3, 3, 128)         73856

 flatten (Flatten)           (None, 1152)              0

 dense (Dense)               (None, 128)               147584

 dropout_2 (Dropout)         (None, 128)               0

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 241,546
Trainable params: 241,546
Non-trainable params: 0
_________________________________________________________________
```

```
# Compile and Train the Model
# After defining the model, we will compile it and train it on the training data.

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

history = model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))

# 1875 is a number of batches. By default batches contain 32 samles.60000 / 32 = 1875
```

```
Epoch 1/10
1875/1875 [==============================] - 90s 47ms/step - loss: 0.5626 - accuracy: 0.7925 - val_loss: 0.4038 - val_accuracy: 0.8
Epoch 2/10
1875/1875 [==============================] - 86s 46ms/step - loss: 0.3651 - accuracy: 0.8662 - val_loss: 0.3155 - val_accuracy: 0.8
Epoch 3/10
1875/1875 [==============================] - 85s 45ms/step - loss: 0.3176 - accuracy: 0.8838 - val_loss: 0.3183 - val_accuracy: 0.8
Epoch 4/10
1875/1875 [==============================] - 85s 45ms/step - loss: 0.2941 - accuracy: 0.8930 - val_loss: 0.2715 - val_accuracy: 0.8
Epoch 5/10
1875/1875 [==============================] - 83s 44ms/step - loss: 0.2765 - accuracy: 0.8981 - val_loss: 0.2725 - val_accuracy: 0.9
Epoch 6/10
1875/1875 [==============================] - 85s 45ms/step - loss: 0.2641 - accuracy: 0.9026 - val_loss: 0.2562 - val_accuracy: 0.9
Epoch 7/10
1875/1875 [==============================] - 83s 44ms/step - loss: 0.2506 - accuracy: 0.9070 - val_loss: 0.2545 - val_accuracy: 0.9
Epoch 8/10
1875/1875 [==============================] - 82s 43ms/step - loss: 0.2425 - accuracy: 0.9099 - val_loss: 0.2562 - val_accuracy: 0.9
Epoch 9/10
1875/1875 [==============================] - 81s 43ms/step - loss: 0.2327 - accuracy: 0.9134 - val_loss: 0.2791 - val_accuracy: 0.8
Epoch 10/10
1875/1875 [==============================] - 83s 44ms/step - loss: 0.2284 - accuracy: 0.9145 - val_loss: 0.2539 - val_accuracy: 0.9
```

```
# Finally, we will evaluate the performance of the model on the test data.

test_loss, test_acc = model.evaluate(x_test, y_test)
```

```
print('Test accuracy:', test_acc)
```

```
313/313 [==============================] - 3s 10ms/step - loss: 0.2606 - accuracy: 0.9031
Test accuracy: 0.9031000137329102
```