**PRACTICAL 1**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Semester:** | VI | **Division:** | 6 |
| **Roll No.:** | 21BCP359 | **Date:** | 10-01-24 | **Batch:** | G11 |
| **Aim:** | WAP to implement DFS and BFS for traversing a graph from source node (S) to goal node (G), where source node and goal node is given by the user as an input. | | | | |

**Program**

from collections import deque

import timeit

*def calculate\_distance\_bfs(graph, path, state, end):*

visited = set()

distance = 0

count = 0

while state != end:

for key in list((graph[state]).keys()):

if key not in visited:

distance = distance + graph[state][key]

visited.add(key)

if key == end:

break

if key == end:

break

count = count + 1

state = path[count]

return distance

*def tsp\_bfs(graph, start, end):*

visited = set()

path = []

distance = 0

queue = deque([start])

visited.add(start)

while queue:

vertex = queue.popleft()

path.append(vertex)

if vertex == end:

distance = calculate\_distance\_bfs(graph, path, start, end)

return path, distance

for adj in graph[vertex]:

if adj not in visited:

visited.add(adj)

queue.append(adj)

return path, distance

*def calculate\_distance\_dfs(graph, path):*

distance = 0

for i in range(len(path) - 1):

distance += graph[path[i]][path[i + 1]]

return distance

*def tsp\_dfs(graph, start, stop):*

visited = set()

stack = [start]

path = []

distance = 0

while stack:

vertex = stack.pop()

path.append(vertex)

visited.add(vertex)

if vertex == stop:

distance = calculate\_distance\_dfs(graph, path)

return path, distance

temp\_stack = []

for adj in graph[vertex]:

if adj not in visited:

temp\_stack.append(adj)

stack.extend(temp\_stack[::-1])

return path, distance

*if \_\_name\_\_ == "\_\_main\_\_":*

graph\_1 = {

"A": {"B": 22, "C": 48, "D": 28},

"B": {"A": 22, "C": 20, "D": 18},

"C": {"A": 48, "B": 20, "D": 32},

"D": {"A": 28, "B": 18, "C": 32},

}

graph\_2 = {

"A": {"B": 2, "G": 6},

"B": {"A": 2, "C": 7, "E": 2},

"C": {"B": 7, "D": 3, "F": 3},

"D": {"C": 3, "H": 2},

"E": {"B": 2, "F": 2, "G": 1},

"F": {"C": 3, "E": 2, "H": 2},

"G": {"A": 6, "E": 1, "H": 4},

"H": {"D": 2, "F": 2, "G": 4},

}

start = input("Enter the starting node: ")

end = input("Enter the ending node: ")

***# DFS***

start\_time\_dfs = timeit.default\_timer()

path, dist = tsp\_dfs(graph\_2, start, end)

execution\_time\_dfs = timeit.default\_timer() - start\_time\_dfs

***# BFS***

start\_time\_bfs = timeit.default\_timer()

path, dist = tsp\_bfs(graph\_2, start, end)

execution\_time\_bfs = timeit.default\_timer() - start\_time\_bfs

print("\nDFS Path:", "".join(path))

print("DFS Cost:", dist)

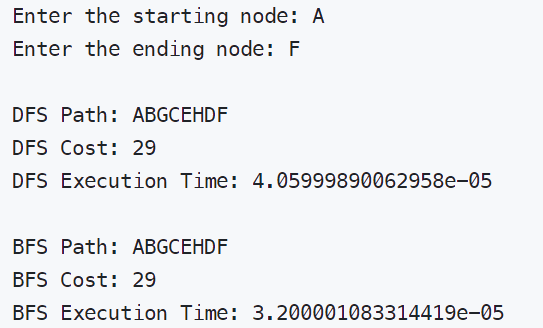
print("DFS Execution Time:", execution\_time\_dfs)

print("\nBFS Path:", "".join(path))

print("BFS Cost:", dist)

print("BFS Execution Time:", execution\_time\_bfs)

**Output**



**Results**

According to the results the **DFS** traversing **takes more time** than **BFS** traversing. Hence in the given example BFS outperforms DFS.

**PRACTICAL 2**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Semester:** | VI | **Division:** | 6 |
| **Roll No.:** | 21BCP359 | **Date:** | 17-01-24 | **Batch:** | G11 |
| **Aim:** | You are given two jugs with m litres and a n litre capacity. Both the jugs are initially empty. The jugs don’t have markings to allow measuring smaller quantities. You have to use the jugs to measure d litres of water where d is less than n. | | | | |

**Program**

from collections import deque

def water\_jug\_BFS(x, y, z):

    visited = set()

    queue = deque([((0, 0), [])])

    while queue:

        (jug\_a, jug\_b), actions = queue.popleft()

        if jug\_a == z or jug\_b == z or jug\_a + jug\_b == z:

            return actions + ["Success"], True

        if (jug\_a, jug\_b) in visited:

            continue

        visited.add((jug\_a, jug\_b))

        # Fill jug A

        if jug\_a < x:

            queue.append(((x, jug\_b), actions + ["Fill A"]))

        # Fill jug B

        if jug\_b < y:

            queue.append(((jug\_a, y), actions + ["Fill B"]))

        # Empty jug A

        if jug\_a > 0:

            queue.append(((0, jug\_b), actions + ["Empty A"]))

        # Empty jug B

        if jug\_b > 0:

            queue.append(((jug\_a, 0), actions + ["Empty B"]))

        # Pour from A to B

        if jug\_a + jug\_b >= y:

            queue.append(((jug\_a - (y - jug\_b), y), actions + ["Pour A to B"]))

        else:

            queue.append(((0, jug\_a + jug\_b), actions + ["Pour A to B"]))

        # Pour from B to A

        if jug\_a + jug\_b >= x:

            queue.append(((x, jug\_b - (x - jug\_a)), actions + ["Pour B to A"]))

        else:

            queue.append(((jug\_a + jug\_b, 0), actions + ["Pour B to A"]))

    return [], False

if \_\_name\_\_ == "\_\_main\_\_":

    n = int(input("Enter jug A's capacity (n): "))

    m = int(input("Enter jug B's capacity (m): "))

    d = int(input("Enter capacity to measure (d): "))

    actions, result = water\_jug\_BFS(n, m, d)

    if result:

        print("The sequence of actions is:")

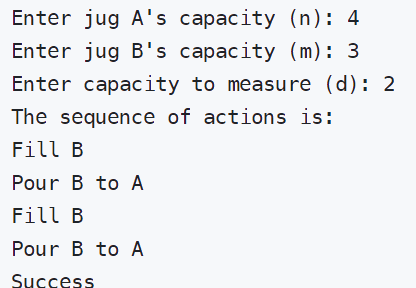
        for action in actions:

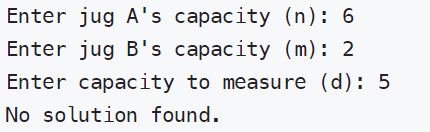
            print(action)

    else:

        print("No solution found.")

**Output**





**PRACTICAL 3**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Semester:** | VI | **Division:** | 6 |
| **Roll No.:** | 21BCP359 | **Date:** | 24-01-24 | **Batch:** | G11 |
| **Aim:** | Solve 8 puzzle problem using A\* algorithm where initial state and Goal state will be given by the users. | | | | |

**Program**

import numpy as np

*# Function to get matrix input from the user*

def get\_matrix\_input(*prompt*):

    print(prompt)

    matrix = []

    for i in range(3):

*# Get each row of the matrix from the user*

        row = list(

            map(

                int,

                input(

                    "Enter row {} (separate numbers with space): ".format(i + 1)

                ).split(),

            )

        )

        matrix.append(row)

    return np.array(matrix)

*# Function to calculate heuristic of a matrix*

def heuristic(*matrix*, *end\_matrix*):

*# Compare each element of the matrix with the end matrix*

    res = matrix == end\_matrix

*# Return the number of elements that are not in their correct position*

    return 9 - np.count\_nonzero(res)

*# Function to generate possible children of a matrix*

def possibleChildren(*matrix*, *e\_matrix*):

    visited.append(matrix)

    [i], [j] = np.where(matrix == 0)  *# Find the position of the empty space (0)*

    direction = [

        [-1, 0],

        [0, -1],

        [1, 0],

        [0, 1],

    ]  *# Possible directions to move the empty space*

    children = []

    for dir in direction:

        ni = i + dir[0]

        nj = j + dir[1]

        newMatrix = matrix.copy()

*# Check if the move is within the bounds of the matrix*

        if ni >= 0 and ni <= 2 and nj >= 0 and nj <= 2:

*# Swap the empty space with the adjacent element*

            newMatrix[i, j], newMatrix[ni, nj] = matrix[ni, nj], matrix[i, j]

*# Check if the new matrix has been visited before*

            if not (any(np.array\_equal(newMatrix, i) for i in visited)):

                visited.append(newMatrix)

                newMatrix\_heu = heuristic(newMatrix, end\_matrix)

                children.append([newMatrix\_heu, newMatrix])

*# Sort the children based on their heuristic*

    children = sorted(children, *key*=lambda *x*: x[0])

    for i in range(len(children)):

        children[i] = children[i][1]

    return children

*# Function to solve the 8-puzzle problem using A\* algorithm*

def a\_star\_8\_puzzle(*start\_matrix*, *end\_matrix*):

    start\_heuristic = heuristic(start\_matrix, end\_matrix)

    if start\_heuristic == 0:

        for node in closed:

            print(node)

        return True

    else:

        children = possibleChildren(start\_matrix, end\_matrix)

        if len(children) > 0:

            for i in range(len(children)):

                open.insert(i, children[i])

        if len(open) > 0:

            newHeu = heuristic(open[0], end\_matrix)

            newMatrix = open[0]

            closed.append(open[0])

            open.pop(0)

            if newHeu == 0:

                for node in closed:

                    print(node)

                return True

            else:

                a\_star\_8\_puzzle(newMatrix, end\_matrix)

        else:

            return False

*# Get the start and end matrices from the user*

start\_matrix = get\_matrix\_input("Enter the start matrix:")

end\_matrix = get\_matrix\_input("Enter the end matrix:")

visited = []

open = []

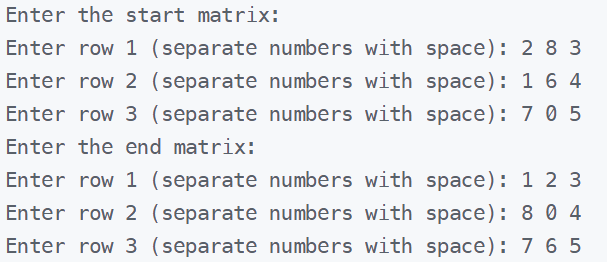
closed = []

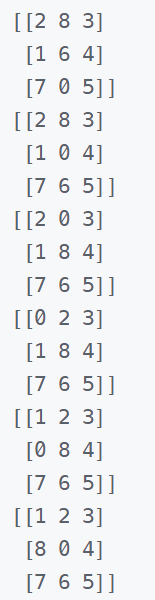
closed.append(start\_matrix)

if \_\_name\_\_ == "\_\_main\_\_":

    a\_star\_8\_puzzle(start\_matrix, end\_matrix)

**Output**





**PRACTICAL 4**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Semester:** | VI | **Division:** | 6 |
| **Roll No.:** | 21BCP359 | **Date:** | 31-01-24 | **Batch:** | G11 |
| **Aim:** | Implement the Fixed Increment Perceptron Learning algorithm as presented in the attachment. The training set for a 2- classification problem is also attached.  Iterate the perceptron through the training set and obtain the weights. | | | | |

**Program**

#include <bits/stdc++.h>

using namespace std;

pair<int, int> input\_dimensions = {8, 2};

vector<pair<double, double>> X = {{0.32, 0.41}, {0.27, 0.54}, {0.57, 0.42}, {0.59, 0.71}, {0.78, 0.82}, {0.79, 0.95}, {1.0, 0.85}, {1.0, 1.1}};

vector<double> coefficients, results, hidden\_values, bias;

vector<double> targets = {0, 1, 0, 1, 0, 1, 0, 1};

void initialize\_coefficients()

{

    uniform\_real\_distribution<double> unif(-0.00002, 0.00002);

    default\_random\_engine re;

    for (int i = 0; i < input\_dimensions.second; i++)

    {

        coefficients.push\_back(unif(re));

    }

    bias.push\_back(unif(re));

}

void calculate\_hidden\_values()

{

    double res1, sum;

    sum = 0;

    hidden\_values.clear();

    for (int i = 0; i < input\_dimensions.first; i++)

    {

        res1 = (X[i].first) \* (coefficients[0]) + (X[i].second) \* (coefficients[1]);

        hidden\_values.push\_back((res1 + bias[0]));

    }

}

void make\_predictions()

{

    calculate\_hidden\_values();

    results.clear();

    for (int i = 0; i < input\_dimensions.first; i++)

    {

        if (hidden\_values[i] > 0)

        {

            results.push\_back(1);

        }

        else

        {

            results.push\_back(0);

        }

    }

}

double evaluate\_accuracy()

{

    double acc = 0;

    for (int i = 0; i < input\_dimensions.first; i++)

    {

        if (results[i] == targets[i])

        {

            acc++;

        }

    }

    return acc / input\_dimensions.first;

}

void adjust\_coefficients(double *learning\_rate*)

{

    double error1, error2;

    error1 = 0;

    error2 = 0;

    for (int i = 0; i < input\_dimensions.first; i++)

    {

        error1 += (targets[i] - results[i]) \* X[i].first;

        error2 += (targets[i] - results[i]) \* X[i].second;

        bias[0] += learning\_rate \* (targets[i] - results[i]);

    }

    coefficients[0] += learning\_rate \* error1;

    coefficients[1] += learning\_rate \* error2;

}

int main()

{

    double learning\_rate, accuracy;

    learning\_rate = 0.000000001;

    initialize\_coefficients();

    make\_predictions();

    accuracy = evaluate\_accuracy();

    long long ep = 1;

    while (accuracy < 1)

    {

        adjust\_coefficients(learning\_rate);

        make\_predictions();

        accuracy = evaluate\_accuracy();

*// cout << "\tepoch: " << ep << "\taccuracy: " << accuracy << "\n";*

        ep++;

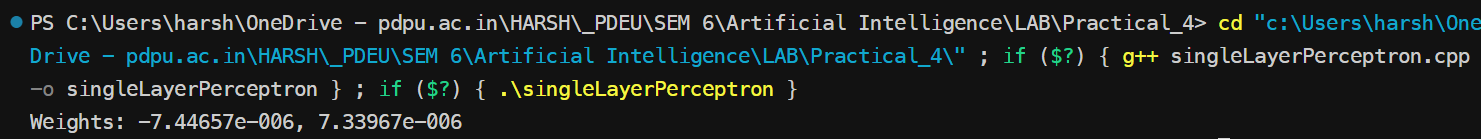
    }

    cout << "Weights: " << coefficients[0] << ", " << coefficients[1] << endl;

    return 0;

}

**Output**



**PRACTICAL 5**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Semester:** | VI | **Division:** | 6 |
| **Roll No.:** | 21BCP359 | **Date:** | 07-02-24 | **Batch:** | G11 |
| **Aim:** | Given a C++ code, identify the algorithm implemented through the code. Also document the code. | | | | |

**Program**

#include <iostream>

#include <cstdio>

#include <cstdlib>

#include <cstring>

#include <ctime>

#include <cmath>

using namespace std;

// GLOBAL FILE NAME

char file\_name[9], file\_name\_inf[14], file\_name\_wgt[14], file\_name\_rst[14];

char file\_name\_out[14], file\_name\_dat[14];

// Class representing a matrix

class matrix {

int row, col;

public:

float mat[15][15];

matrix() {

row = 0;

col = 0;

}

void set(int, int);

int getrows() {

return row;

}

int getcols() {

return col;

}

void getdata();

FILE \*fgetdata(FILE \*);

void displaydata();

void displaydat();

FILE \*fputdata(FILE \*);

FILE \*fputdat(FILE \*);

matrix operator+(matrix);

matrix operator-();

matrix operator\*(matrix);

matrix operator\*(float);

};

// Set the dimensions of the matrix

void matrix::set(int i, int j) {

row = i;

col = j;

}

// Read matrix data from file

FILE \*matrix::fgetdata(FILE \*fmat) {

char line;

int i, j;

fscanf(fmat, "%d%d", &(row), &(col));

for (i = 1; i <= row; i++)

for (j = 1; j <= col; j++)

fscanf(fmat, "%f", &(mat[i][j]));

return (fmat);

}

// Read matrix data from user input

void matrix::getdata() {

int i, j;

cout << "Enter the size of the matrix:";

cin >> row >> col;

for (i = 1; i <= row; i++)

for (j = 1; j <= col; j++) {

cout << "element [" << i << "] [ " << j << " ] ";

cin >> mat[i][j];

}

}

// Display matrix data

void matrix::displaydata() {

int i, j;

for (i = 1; i <= row; i++, printf("\n\r"))

for (j = 1; j <= col; j++, printf("\t"))

printf("\t\t%10.2f", mat[i][j]);

}

// Display matrix dimensions

void matrix::displaydat() {

int i;

cout << row;

}

// Write matrix data to file

FILE \*matrix::fputdata(FILE \*fmat) {

int i, j;

fprintf(fmat, "%d\n%d\n", row, col);

for (i = 1; i <= row; i++)

for (j = 1; j <= col; j++)

fprintf(fmat, "%f\n", mat[i][j]);

return (fmat);

}

// Write matrix dimensions to file

FILE \*matrix::fputdat(FILE \*fmat) {

int i;

fprintf(fmat, "%d", row);

return (fmat);

}

// Overloaded operator for matrix addition

matrix matrix::operator+(matrix m) {

matrix temp;

int i, j;

if ((row == m.row) && (col == m.col))

for (i = 1; i <= row; i++)

for (j = 1; j <= col; j++)

temp.mat[i][j] = mat[i][j] + m.mat[i][j];

else {

cout << "The addition of the matrices is not possible";

exit(1);

}

temp.row = row;

temp.col = col;

return (temp);

}

// Overloaded operator for matrix transposition

matrix matrix::operator-() {

matrix temp;

int i, j;

temp.row = col;

temp.col = row;

for (i = 1; i <= col; i++)

for (j = 1; j <= row; j++)

temp.mat[i][j] = mat[j][i];

return (temp);

}

// Overloaded operator for matrix multiplication

matrix matrix::operator\*(matrix m) {

matrix temp;

int i, j, k;

if (col == m.row) {

for (i = 1; i <= row; i++)

for (j = 1; j <= m.col; j++) {

temp.mat[i][j] = 0;

for (k = 1; k <= col; k++)

temp.mat[i][j] = temp.mat[i][j] + (mat[i][k] \* m.mat[k][j]);

}

} else {

cout << "The multiplication of the matrices is not possible";

exit(1);

}

temp.row = row;

temp.col = m.col;

return (temp);

}

// Overloaded operator for scalar multiplication with matrix

matrix matrix::operator\*(float svalue) {

matrix temp;

int i, j;

for (i = 1; i <= row; i++)

for (j = 1; j <= col; j++)

temp.mat[i][j] = mat[i][j] \* svalue;

temp.row = row;

temp.col = col;

return (temp);

}

// Class representing the training process of the neural network

class training {

FILE \*fin, \*fout, \*fwt;

matrix Input[5], Output[5], Weights[5], dWeights[5], d, e, T;

float alpha, eta, err, theta, lamda, error;

int TotalLayers, HiddenLayers, l[5], ntest, iterates;

long filepos;

public:

training();

void readinputs();

void printing();

void initweights();

void initdweights();

void train();

void io\_values();

void backpropagate();

void errors();

void chgweights();

void newweights();

~training();

};

// Constructor for training class

training::training() {

// Open files for input and output

if ((fin = fopen(file\_name\_dat, "r")) == NULL) exit(1);

if ((fout = fopen(file\_name\_out, "w")) == NULL) exit(1);

if ((fwt = fopen(file\_name\_wgt, "w")) == NULL) exit(1);

readinputs();

printing();

initweights();

initdweights();

train();

}

// Function to read input parameters for training

void training::readinputs() {

int i;

error = 0;

char line;

fscanf(fin, "%d", &HiddenLayers); // Get number of hidden layers

TotalLayers = HiddenLayers + 1; // Calculate total number of layers

for (i = 0; i <= TotalLayers; i++)

fscanf(fin, "%d", &l[i]);

fscanf(fin, "%f%f%f%f%f", &alpha, &err, &eta, &theta, &lamda);

fscanf(fin, "%d%d", &ntest, &iterates);

filepos = ftell(fin);

}

// Function to print input parameters for training

void training::printing() {

// Print parameters to output file

for (int i = 0; i <= TotalLayers; i++)

fprintf(fout, "\nNumber of Neurons in layer[%d]=%d", i + 1, l[i]);

fprintf(fout, "\nAlpha value(Momentum factor): %f", alpha);

fprintf(fout, "\nError constant : %f", err);

fprintf(fout, "\nLearning rate : %f", eta);

fprintf(fout, "\nThreshold value : %f", theta);

fprintf(fout, "\nScaling Parameter: %f", lamda);

fprintf(fout, "\nNo of Training data : %d", ntest);

fprintf(fout, "\nMaximum Iteration : %d", iterates);

system("cls");

// Print parameters to console

printf("\n\n\n");

for (int i = 0; i <= TotalLayers; i++)

printf("\n\t\tNumber of Neurons in layer[%d]=%d", i + 1, l[i]);

printf("\n\n\t\tAlpha value(Momentum factor): %f", alpha);

printf("\n\t\tError constant : %f", err);

printf("\n\t\tLearning rate : %f", eta);

printf("\n\t\tThreshold value : %f", theta);

printf("\n\t\tScaling Parameter : %f", lamda);

printf("\n\t\tNo of Training data : %d", ntest);

printf("\n\t\tMaximum Iteration : %d", iterates);

cin.get();

}

// Function to initialize weights randomly

void training::initweights() {

srand(2000);

srand(time(0));

for (int k = 0; k < TotalLayers; k++) {

Weights[k].set(l[k], l[k + 1]);

for (int i = 1; i <= l[k]; i++)

for (int j = 1; j <= l[k + 1]; j++)

Weights[k].mat[i][j] = ((float)rand() / 32767) - 0.5;

fprintf(fout, "\nWeights[%d]:", k);

Weights[k].fputdata(fout);

}

}

// Function to initialize difference in weights

void training::initdweights() {

for (int k = 0; k < TotalLayers; k++) {

dWeights[k].set(l[k], l[k + 1]);

for (int i = 1; i <= l[k]; i++)

for (int j = 1; j <= l[k + 1]; j++)

dWeights[k].mat[i][j] = 0.0;

}

}

// Function to perform neural network training

void training::train() {

int k;

for (int jtr = 1; jtr <= iterates; jtr++) {

error = 0.0;

fseek(fin, filepos, SEEK\_SET);

cout << "\nIteration Number: " << jtr << endl;

for (int itr = 1; itr <= ntest; itr++) {

Input[0].fgetdata(fin);

T.fgetdata(fin);

cout << "\rTraining Data Number: " << itr;

io\_values();

backpropagate();

errors();

chgweights();

newweights();

}

fprintf(fout, " %10.3E\n", error / ntest);

}

cin.get();

for (k = 0; k < TotalLayers; k++)

fwt = Weights[k].fputdata(fwt);

}

// Function to calculate input/output values of neurons

void training::io\_values() {

Output[0] = Input[0];

for (int m = 0; m <= TotalLayers - 1; m++) {

Input[m + 1] = -Weights[m] \* Output[m];

Output[m + 1].set(l[m + 1], 1);

for (int i = 1; i <= l[m + 1]; i++)

Output[m + 1].mat[i][1] = 1.0 / (1.0 + exp(-lamda \* (Input[m + 1].mat[i][1] + theta)));

}

}

// Function to perform backpropagation

void training::backpropagate() {

d.set(l[TotalLayers], 1);

for (int i = 1; i <= l[TotalLayers]; i++)

d.mat[i][1] = Output[TotalLayers].mat[i][1] \* (1 - Output[TotalLayers].mat[i][1]) \* (T.mat[i][1] - Output[TotalLayers].mat[i][1]);

dWeights[TotalLayers - 1] = (dWeights[TotalLayers - 1] \* alpha) + ((Output[TotalLayers - 1] \* -d) \* eta);

}

// Function to calculate errors

void training::errors() {

float sum = 0.0, x, y1, y2;

for (int j = 1; j <= l[TotalLayers]; j++) {

y1 = T.mat[j][1];

y2 = Output[TotalLayers].mat[j][1];

x = fabs(y1 - y2);

x = x \* x;

sum = sum + x;

}

sum = sqrt(sum / l[TotalLayers]);

error = error + sum;

cout << "\t\t Error =" << error;

}

// Function to calculate change in weights

void training::chgweights() {

int k;

for (int i = 0; i <= TotalLayers - 2; i++) {

k = TotalLayers - i - 1;

e = Weights[k] \* d;

d.set(l[k], 1);

for (int j = 1; j <= l[k]; j++) {

d.mat[j][1] = Output[k].mat[j][1] \* (1 - Output[k].mat[j][1]) \* e.mat[j][1];

}

dWeights[k - 1] = (dWeights[k - 1] \* alpha) + ((Output[k - 1] \* -d) \* eta);

}

}

// Function to update weights

void training::newweights() {

for (int k = 0; k < TotalLayers; k++)

Weights[k] = Weights[k] + dWeights[k];

}

// Destructor for training class

training::~training() {

fclose(fin);

fclose(fout);

fclose(fwt);

}

// Class representing the inference process of the neural network

class inference {

FILE \*fin, \*fout, \*fwt;

matrix Input[5], Output[5], Weights[5], T, CalculatedErr, NoOfTest;

float alpha, eta, err, theta, x1, x2, lamda, Calerror;

int TotalLayers, ntest, l[10];

public:

inference();

void readinputs();

void initweights();

void i\_values();

void calculate();

void error();

~inference();

};

// Constructor for inference class

inference::inference() {

// Open files for input and output

if ((fin = fopen(file\_name\_inf, "r")) == NULL) exit(1);

if ((fout = fopen(file\_name\_rst, "w")) == NULL) exit(1);

if ((fwt = fopen(file\_name\_wgt, "r")) == NULL) exit(1);

readinputs();

initweights();

i\_values();

calculate();

error();

}

// Function to read input parameters for inference

void inference::readinputs() {

int i;

fscanf(fin, "%d", &TotalLayers); // Get number of hidden layers

for (i = 0; i <= TotalLayers; i++)

fscanf(fin, "%d", &l[i]); // Get number of neurons in each layer

fscanf(fin, "%f%f%f%f%f", &alpha, &err, &eta, &theta, &lamda); // Get other parameters

fscanf(fin, "%d", &ntest); // Get number of test cases

}

// Function to initialize weights for inference

void inference::initweights() {

for (int k = 0; k < TotalLayers; k++) {

Weights[k].fgetdata(fwt);

}

}

// Function to calculate input values for inference

void inference::i\_values() {

for (int itr = 1; itr <= ntest; itr++) {

Input[0].fgetdata(fin);

cout << "\rTesting Data Number: " << itr;

Output[0] = Input[0];

for (int m = 0; m <= TotalLayers - 1; m++) {

Input[m + 1] = -Weights[m] \* Output[m];

Output[m + 1].set(l[m + 1], 1);

for (int i = 1; i <= l[m + 1]; i++)

Output[m + 1].mat[i][1] = 1.0 / (1.0 + exp(-lamda \* (Input[m + 1].mat[i][1] + theta)));

}

Output[TotalLayers].displaydat();

}

}

// Function to perform calculation for inference

void inference::calculate() {

float sum = 0.0;

for (int i = 1; i <= ntest; i++) {

T.fgetdata(fin);

CalculatedErr = Output[TotalLayers] - T;

CalculatedErr = -CalculatedErr;

CalculatedErr = CalculatedErr \* CalculatedErr;

x1 = CalculatedErr.mat[1][1];

sum = sum + x1;

}

sum = sqrt(sum / ntest);

Calerror = sum;

}

// Function to calculate error for inference

void inference::error() {

printf("\nCalculated Error: %f", Calerror);

fprintf(fout, "%f", Calerror);

}

// Destructor for inference class

inference::~inference() {

fclose(fin);

fclose(fout);

}

// Main function

int main() {

strcpy(file\_name, "Nndat.dat");

strcpy(file\_name\_inf, "Nntst.dat");

strcpy(file\_name\_wgt, "Nnwgt.dat");

strcpy(file\_name\_out, "Nnout.dat");

strcpy(file\_name\_rst, "Nnres.dat");

training mlp1;

inference mlp2;

return 0;

}

**Explanation**

The algorithm implemented through the code is **Backpropagation for training a Multi-Layer Perceptron (MLP)** neural network.

Here's a breakdown of the code and its functionalities:

**Classes:**

* matrix: This class represents a matrix and provides methods for creating, manipulating, and displaying matrices.
* training: This class handles the training process of the MLP network. It includes methods for reading training data, initializing weights, performing backpropagation, calculating errors, and updating weights.
* inference: This class performs inference on the trained network. It reads test data, calculates the network's output, and compares it to the desired output.

**Training Process (training class):**

1. **Initialization:**

* Reads training data and network configuration from a file.
* Initializes weights and learning parameters.

1. **Iteration Loop:**

* Loops for a specified number of iterations.
* For each iteration:
* Loops for each training data point:
* Calculates the output of each layer using the forward pass.
* Performs backpropagation to calculate the error gradients.
* Updates the weights using the gradient descent algorithm with momentum.

1. **Weight Update:**

* Writes the final weights to a file.

**Inference Process (inference class):**

1. **Loading Configuration:**

* Reads network configuration and weights from files.

1. **Test Data Loop:**

* Loops for each test data point:
* Calculates the network's output using the forward pass.
* Compares the output to the desired output and calculates the error.
* Writes the calculated output, actual output, and error to a file.

**Overall, the code implements a backpropagation algorithm to train a multilayer perceptron neural network. The trained network can then be used for inference on new data.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Semester:** | VI | **Division:** | 6 |
| **Roll No.:** | 21BCP359 | **Date:** | 14-02-24 | **Batch:** | G11 |
| **Aim:** | **Part 1:**  Understand the project available on following link  Project Link: <https://github.com/aharley/nn_vis>  Project by: <https://adamharley.com/>  Reference in case needed: <https://www.youtube.com/watch?v=pj9-rr1wDhM>  **Part 2:**  Populate the table below to summarize your understanding of the project mentioned in part 1 | | | | |

**PRACTICAL 6**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Layer** | **Task** | **Rationale** | | Input layer | It receives user input and converts raw pixels from a sketchpad into data that the system can process. | To take raw input from the user and preprocess it. | | Convolutional layer | It identifies patterns in the input data, like edges and corners, by applying mathematical operations and activation functions. | Extracts features, like edges and corners. | | Pooling layer | It reduces the size of the data while keeping important information intact, making computations more efficient. It does this by condensing features and focusing on the most significant values. | To reduce space of the matrix (reduce spatial dimensions of feature maps) while conserving the original image. Consider the pixel having the highest value (illumination) using a stride of 2\*2 pixels (2\*2 max pooling) and taking that value to just one pixel in the new matrix. | | Classifying layer | It utilizes the extracted features to accurately categorize the input data. It consists of interconnected neurons that analyze the features for classification. | The classifying layer takes the high-level abstracted features from previous layers and uses them to classify input data into different categories. There are 120 neurons in the first layer and 100 neurons in the second. | | Output layer | It generates the final prediction based on the classification results, with each neuron representing the probability of a specific outcome, such as recognizing different digits. | Produces the final output or prediction of the network, representing the class probabilities. | | |
| **How does the following hyper-parameters affect the network performance**   |  |  |  | | --- | --- | --- | | **Hyper-Parameter** | **One Line Definition** | **Effect on the CNN** | | Stride | Determines how much the filter moves across the input image. | Changing the stride impacts the size of the output feature maps. A larger stride means fewer calculations and smaller output maps, speeding up processing. | | Dilation Rate | Controls how the elements of the convolutional filter are spread out. | Increasing dilation rate expands the filter's view, allowing it to capture broader features but at the cost of reduced detail in the output. | | Type of pooling layer | Dictates how feature maps are condensed in pooling layers. | Various types like max pooling or average pooling determine how features are combined, impacting the network's capacity to maintain crucial details while decreasing size. | | Kernel size | Determines the dimensions of the convolutional filters. | Bigger sizes gather more nearby details, enabling the network to learn complex patterns and demanding more computations. Smaller sizes concentrate on finer details but might miss broader patterns. | | padding | Adding extra pixels around the input image. | It influences the size of the output feature maps. Zeropadding keeps the size unchanged, valid padding reduces it, and same padding maintains the input size. | |
|  |
| **References:**  [An Intuitive Explanation of Convolutional Neural Networks – the data science blog (ujjwalkarn.me)](https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/)  [Gentle Dive into Math Behind Convolutional Neural Networks | by Piotr Skalski | Towards Data Science](https://towardsdatascience.com/gentle-dive-into-math-behind-convolutional-neural-networks-79a07dd44cf9)  [Intuitively Understanding Convolutions for Deep Learning | by Irhum Shafkat | Towards Data Science](https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1)  [An Introduction to different Types of Convolutions in Deep Learning | by Paul-Louis Pröve | Towards Data Science](https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d) |

**PRACTICAL 7**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Semester:** | VI | **Division:** | 6 |
| **Roll No.:** | 21BCP359 | **Date:** | 06-03-24 | **Batch:** | G11 |
| **Aim:** | Prepare your version of CNN following the steps in the link shared here”  <https://towardsdatascience.com/build-your-own-convolution-neural-network-in-5-mins-4217c2cf964f> | | | | |

**Implementation:**

import keras

from keras.datasets import mnist

from keras.models import Sequential

from keras.layers import Dense, Dropout, Flatten

from keras.layers import Conv2D, MaxPooling2D

import numpy as np

batch\_size = 128

num\_classes = 10

epochs = 12

*# input image dimensions*

img\_rows, img\_cols = 28, 28

*# the data, split between train and test sets*

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

x\_train = x\_train.reshape(60000, 28, 28, 1)

x\_test = x\_test.reshape(10000, 28, 28, 1)

print("x\_train shape:", x\_train.shape)

print(x\_train.shape[0], "train samples")

print(x\_test.shape[0], "test samples")

*# convert class vectors to binary class matrices*

y\_train = keras.utils.to\_categorical(y\_train, num\_classes)

y\_test = keras.utils.to\_categorical(y\_test, num\_classes)

model = Sequential()

model.add(Conv2D(32, *kernel\_size*=(3, 3), *activation*="relu", *input\_shape*=(28, 28, 1)))

model.add(Conv2D(64, (3, 3), *activation*="relu"))

model.add(MaxPooling2D(*pool\_size*=(2, 2)))

model.add(Dropout(0.25))

model.add(Flatten())

model.add(Dense(128, *activation*="relu"))

model.add(Dropout(0.5))

model.add(Dense(num\_classes, *activation*="softmax"))

model.compile(

*loss*=keras.losses.categorical\_crossentropy,

*optimizer*=keras.optimizers.Adadelta(),

*metrics*=["accuracy"],

)

model.fit(

    x\_train,

    y\_train,

*batch\_size*=batch\_size,

*epochs*=epochs,

*verbose*=1,

*validation\_data*=(x\_test, y\_test),

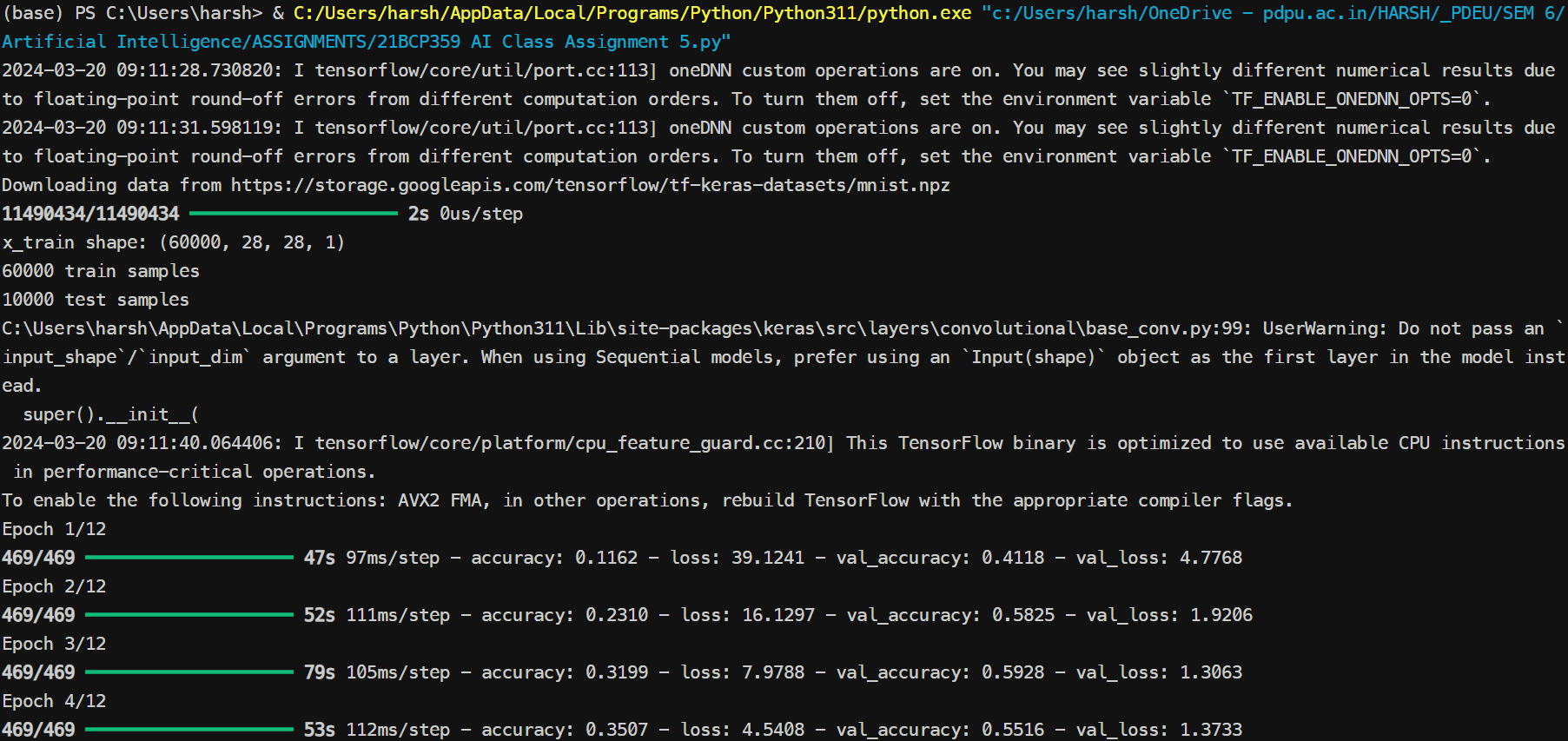
)

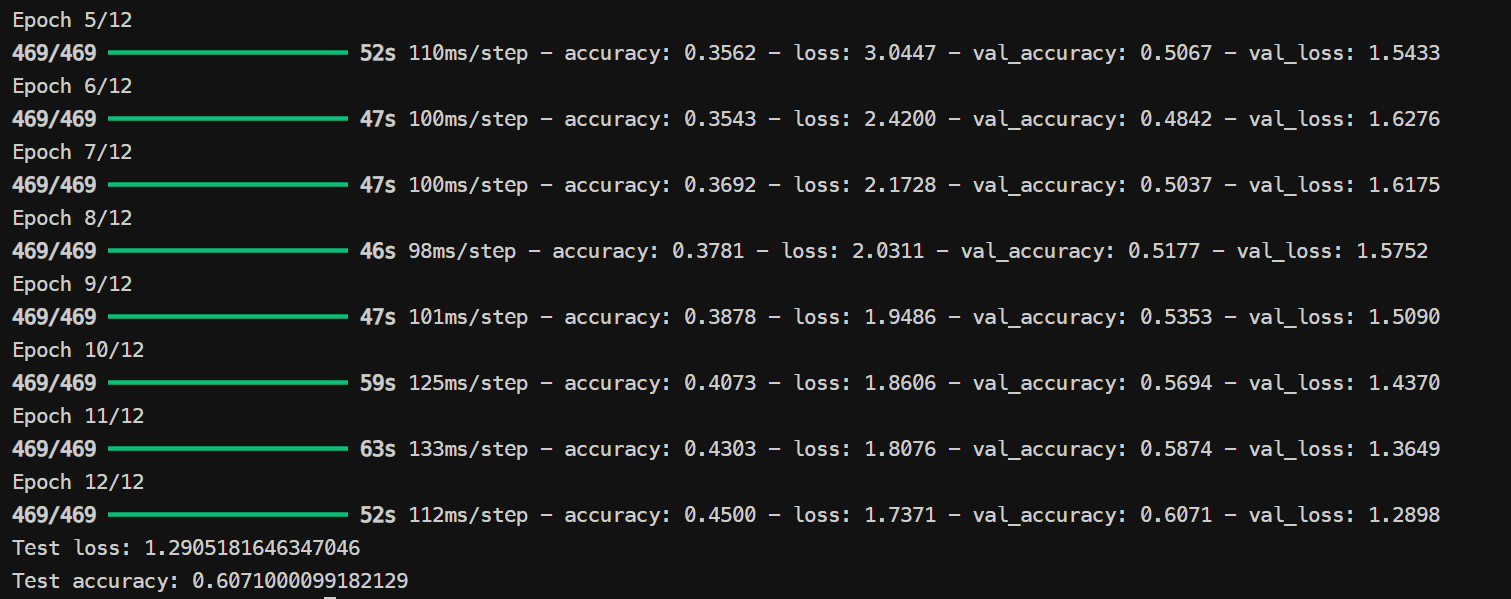
score = model.evaluate(x\_test, y\_test, *verbose*=0)

print("Test loss:", score[0])

print("Test accuracy:", score[1])

**Output**





**PRACTICAL 8**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Semester:** | VI | **Division:** | 6 |
| **Roll No.:** | 21BCP359 | **Date:** | 13-03-24 | **Batch:** | G11 |
| **Aim:** | Design the Neural Network model for the project title submitted by you. Demonstrate **Over-fitting** and solve the same using **Dropout technique**. | | | | |

**Overfitting and Regularization in Plant Disease Detection CNN**

One of the key challenges in training deep learning models, particularly Convolutional Neural Networks (CNNs), is overfitting. Overfitting occurs when a model becomes too specialized on the training data and fails to generalize well to unseen data. This manifests as high training accuracy but low validation accuracy. In the context of plant disease detection, an overfitting model might achieve impressive results on the training images but struggle to accurately diagnose diseases in new images with slightly different characteristics.

To mitigate overfitting and improve the generalizability of our Plant Disease detection CNN model, we can employ a regularization technique called dropout. Dropout randomly drops out a certain percentage of neurons during training. This forces the model to learn redundant representations and prevents it from relying on any single neuron or its specific quirks in the training data. By encountering different "reduced versions" of the training data during each epoch, the model is encouraged to learn more generalizable features that are robust to variations in the data.

**Before Dropout Implementation**

model.add(layers.MaxPooling2D(2, 2))

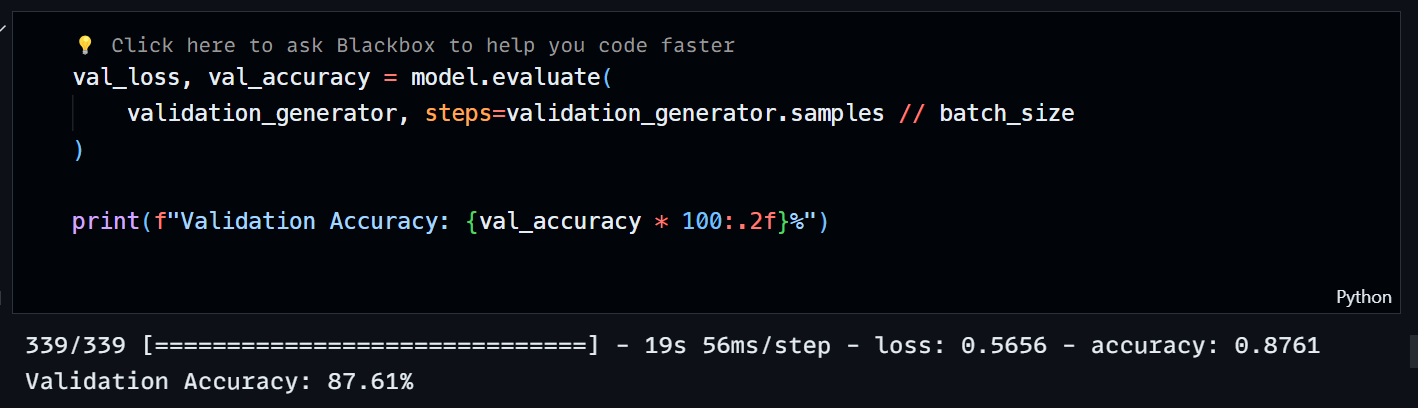
model.add(layers.Conv2D(64, (3, 3), activation="relu"))

model.add(layers.MaxPooling2D(2, 2))

model.add(layers.Flatten())

model.add(layers.Dense(256, activation="relu"))

model.add(layers.Dense(train\_generator.num\_classes, activation="softmax"))



**After Dropout Implementation**

model.add(layers.Conv2D(32, (3, 3), activation="relu", input\_shape=(img\_size, img\_size, 3)))

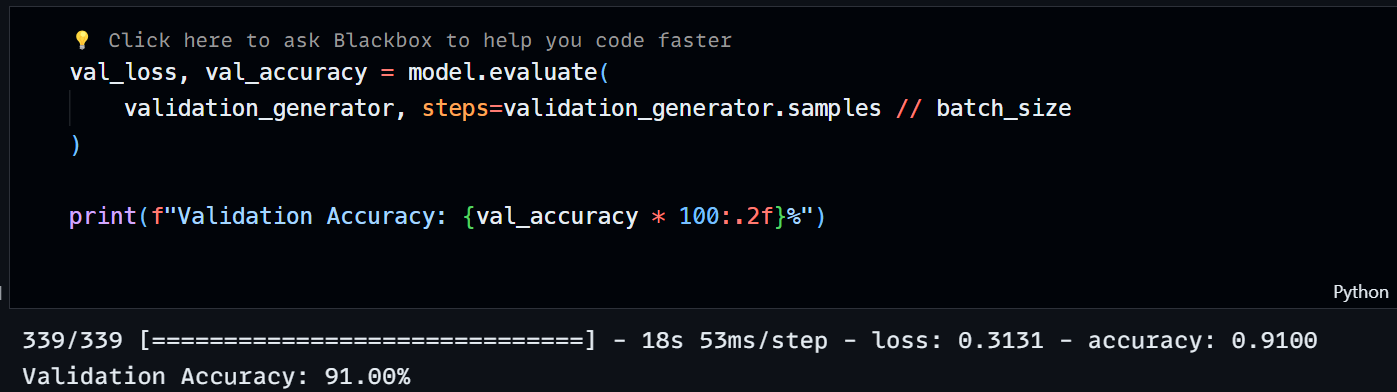
model.add(layers.MaxPooling2D(2, 2))

model.add(layers.Dropout(0.25))

model.add(layers.Conv2D(64, (3, 3), activation="relu"))

model.add(layers.MaxPooling2D(2, 2))

model.add(layers.Dropout(0.25))



In this example, Dropout(0.25) is used after each convolutional layer. This means that during training, 25% of the neurons in those layers will be randomly dropped out. The dropout rate is a hyperparameter that can be tuned for optimal performance. Experimenting with different dropout rates (between 0.2 and 0.5) can help achieve the best balance between preventing overfitting and maintaining model accuracy.

The dropout rate (e.g., 0.25 in the code) determines the percentage of neurons dropped in each layer during training. It's a hyperparameter you can tune. Here's a general guideline:

* **Low Dropout (0-0.2):** Might not be effective enough in preventing overfitting.
* **Moderate Dropout (0.2-0.5):** Often a good starting point for many CNN architectures.
* **High Dropout (0.5-0.8):** Can be too aggressive and might hurt performance if not tuned carefully.

By incorporating dropout as a regularization technique, we can enhance the robustness and generalizability of our Plant Disease detection CNN model. This allows the model to perform more effectively on unseen plant images, ultimately leading to more accurate disease diagnoses.

**PRACTICAL 9**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Semester:** | VI | **Division:** | 6 |
| **Roll No.:** | 21BCP359 | **Date:** | 20-03-2427-03-24 | **Batch:** | G11 |
| **Aim:** | For your project definition demonstrate applicable task out of prediction and classification. Explain the entire work flow of your project through a single diagram. | | | | |

**Plant Disease Classification using CNN**

1. **Data Collection:**

* Gather high-quality images of plant leaves.
* Include images of healthy leaves and leaves with various diseases you aim to classify.
* Ensure the image collection is representative of the target diseases.

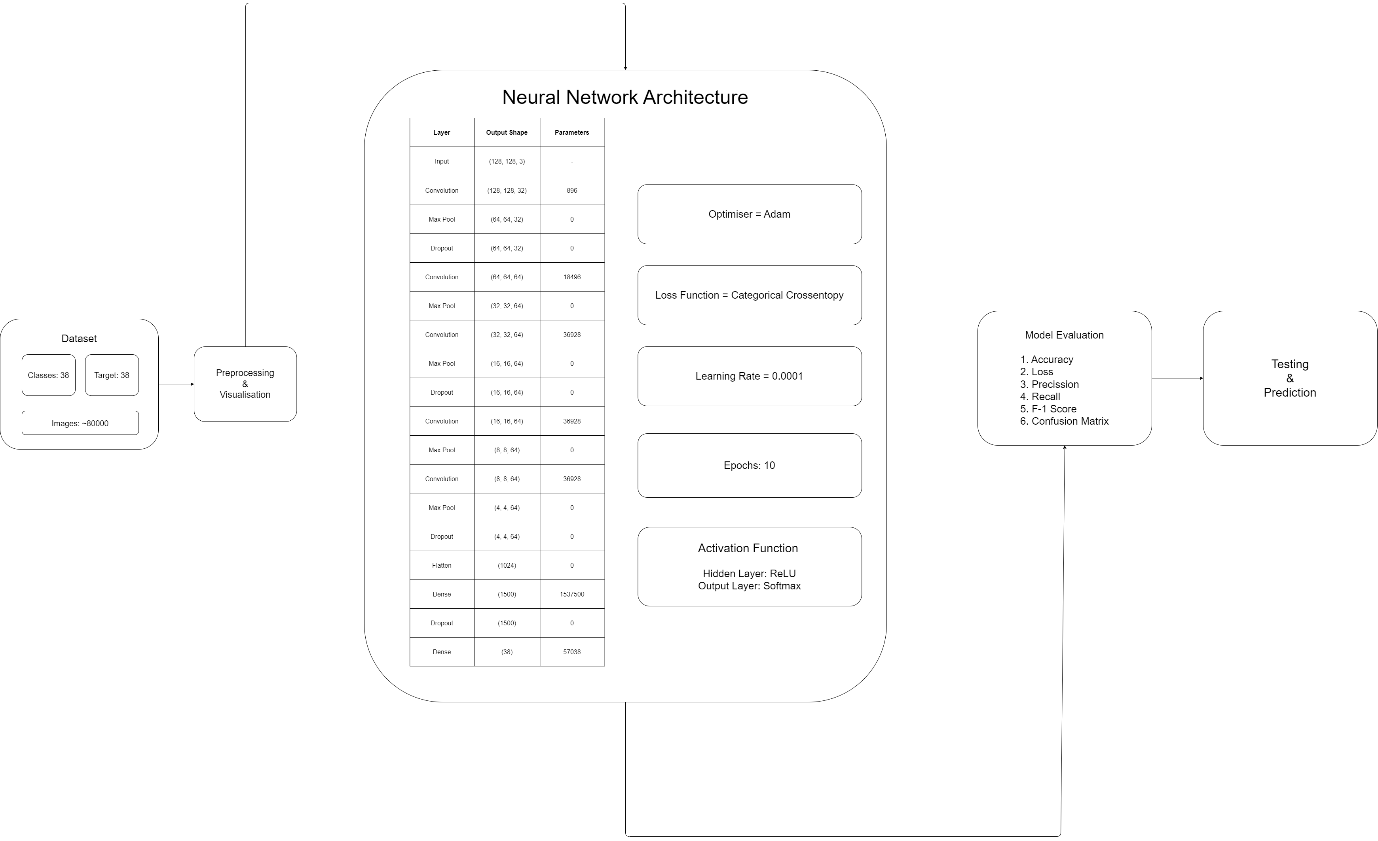
1. **Data Pre-processing:**

* Prepare the images for training the model.
  + 1. **Model Development**
* **Model Architecture:**
  + This model utilizes a Convolutional Neural Network (CNN).
  + CNNs excel at extracting spatial features from images, making them ideal for image classification tasks.
* **Building Blocks:**
  + **Convolutional Layers:** These layers apply filters to the image, identifying patterns and extracting features.
  + **Pooling Layers:** Reduce image dimensionality by summarizing features from previous layers (e.g., Max Pooling).
  + **Activation Functions:** Introduce non-linearity to the network, allowing it to learn complex relationships – ReLU and Softmax.
  + **Flatten Layer:** Transform the extracted features into a one-dimensional vector for feeding into fully-connected layers.
  + **Fully-Connected Layers:** Dense layers that process the flattened features and make the final classification decision.
    1. **Training Process:**
  + Divide the pre-processed data into training, validation, and test sets.
  + The training set is used to train the model, the validation set monitors training progress, and the test set evaluates final model performance on unseen data.
  + The model learns by iteratively adjusting its internal weights based on the training data and the chosen loss function - Categorical Cross entropy for multi-class classification.
  + The optimizer - Adam guides these adjustments to minimize the loss function and improve classification accuracy.

**Evaluation and Deployment**

1. **Model Evaluation:**
   * After training, assess the model's performance on the held-out test set.
   * Common metrics include **accuracy**, **precision**, **recall**, and **F1-score**.
2. **Deployment:**
   * Once satisfied with the model's performance, deploy it for real-world use.
   * This could involve integrating it into a mobile application or web service for on-demand plant disease prediction.

**Flow Chart**

****

**PRACTICAL 10**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Semester:** | VI | **Division:** | 6 |
| **Roll No.:** | 21BCP359 | **Date:** | 27-03-24 | **Batch:** | G11 |
| **Aim:** | For your project demonstrate the following:   * need of optimizer - 5 marks * significance of your choice of optimizer - 5 marks * comparison of outcomes with and without optimization - 5 marks | | | | |

**Need of Optimizer**

In a neural network, an optimizer plays a crucial role in training the model. The primary purpose of an optimizer is to minimise the error between the predicted output of the neural network and the actual target output by adjusting the weights and biases of the network during the training process.

* Error Minimization: Neural networks are trained by adjusting their parameters (weights and biases) to minimise the difference between predicted outputs and actual targets. Optimizers determine how these parameters should be updated to reduce this error effectively.
* Gradient Descent: Most optimizers use some form of gradient descent algorithm to update the weights and biases of the neural network. Gradient descent computes the gradients of the loss function with respect to the network parameters, indicating the direction and magnitude of the steepest decrease in the error. By following these gradients, the optimizer updates the parameters to reach a minimum point in the error surface.
* Speed and Efficiency: Optimizers help in training the neural network efficiently by adjusting the learning rate and other hyperparameters. They prevent the model from getting stuck in local minima and help it converge to the global minimum of the loss function.
* Stability: Some optimizers incorporate techniques to stabilise the training process, such as momentum, adaptive learning rates, and regularisation. These techniques help prevent the model from oscillating or diverging during training.
* Flexibility: Different optimizers have different characteristics and are suitable for different types of problems or architectures. For instance, Adam optimizer is widely used for its adaptive learning rate properties, while SGD (Stochastic Gradient Descent) is simpler and may be more suitable for certain scenarios.

Overall, optimizers are essential components of neural network training, ensuring that the model learns effectively and efficiently from the training data to make accurate predictions on new data.

**Significance of ADAM Optimizer**

Adam (Adaptive Moment Estimation) optimizer is a popular optimization algorithm used in training neural networks. Its significance lies in several key features that make it effective for a wide range of deep learning tasks:

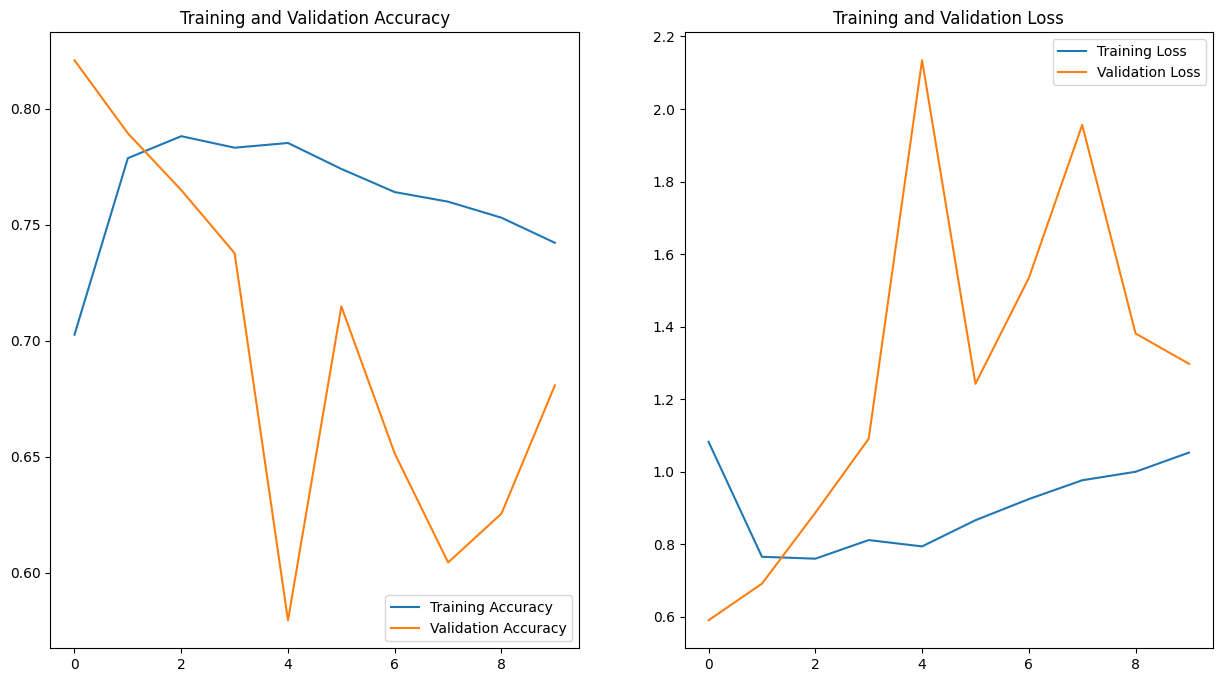
* Adaptive Learning Rates: Adam dynamically adjusts the learning rates for each parameter based on the magnitude of gradients and the past history of gradients for that parameter. This adaptivity helps in faster convergence and efficient training, as it allows larger updates for infrequent parameters and smaller updates for frequent ones.
* Momentum Optimization: Adam incorporates momentum, which helps accelerate convergence by accumulating gradients from past time steps. This helps to navigate through areas of high curvature and reach the minimum of the loss function more efficiently.
* Bias Correction: Adam performs bias correction, particularly in the early stages of training when the estimates of the moments are biased towards zero. This correction helps to improve the stability of training and ensures that the optimization process starts with more accurate estimates.
* Efficient Memory Usage: Adam maintains exponentially decaying averages of past gradients and squared gradients, requiring only first-order moments (mean) and second-order moments (variance). This results in efficient memory usage compared to other optimization algorithms like RMSprop.
* Robustness to Hyperparameters: Adam is relatively less sensitive to hyperparameters compared to other optimization algorithms like SGD. It performs well with default hyperparameters across a wide range of tasks and architectures, making it easier to use for practitioners.
* Widely Used: Adam has become a standard optimizer in many deep learning frameworks and is often the default choice for training neural networks. Its widespread adoption and proven performance on various tasks make it a go-to optimizer for many practitioners.

Overall, the significance of Adam optimizer lies in its ability to provide efficient and effective optimization for neural networks, leading to faster convergence, better generalisation, and easier hyperparameter tuning.

**Comparison with and without Optimizer**

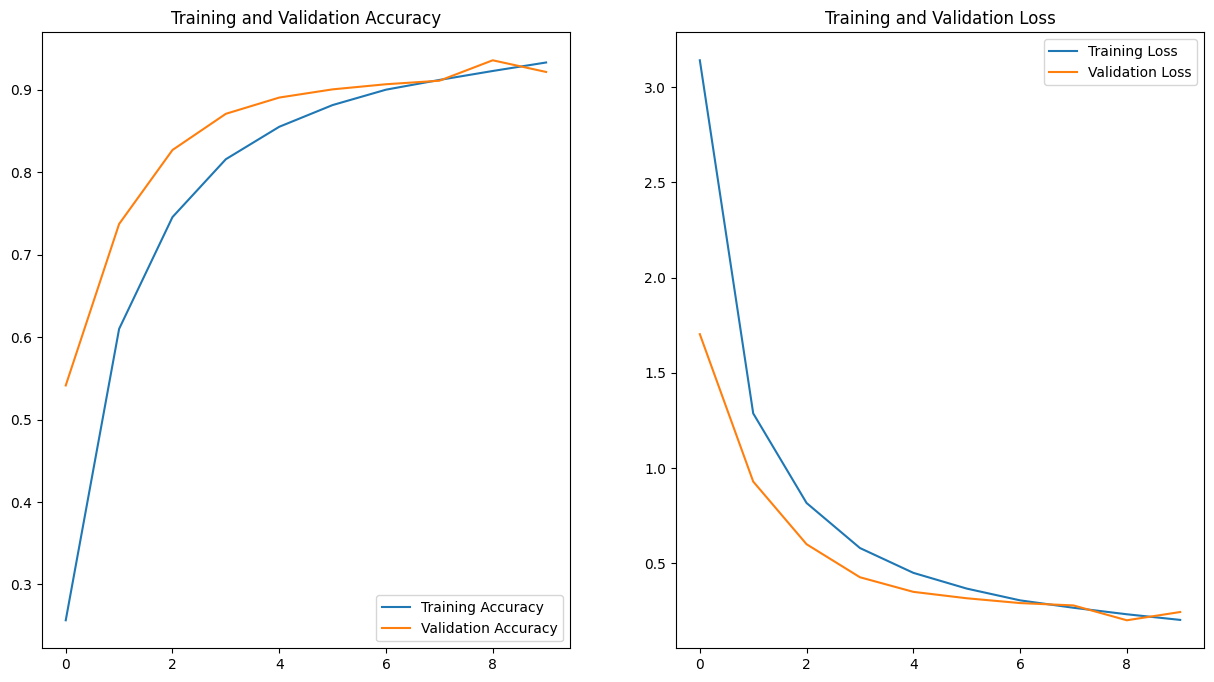
**Without Optimiser**





**With Optimiser**





**PRACTICAL 11**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Semester:** | VI | **Division:** | 6 |
| **Roll No.:** | 21BCP359 | **Date:** | 10-04-24 | **Batch:** | G11 |
| **Aim:** | **11a -** Understanding the basics and IDE for Prolog Programming  **11b -** Implement any two of the following using Prolog:   * Medical diagnosis of common cold and flu using symptom inputs * Demonstrating list in prolog * Monkey banana problem * Find the factorial of a given number | | | | |

**11a: Understanding the basics and IDE for Prolog Programming**

Prolog is a logic programming language associated with artificial intelligence and computational linguistics. In Prolog, programs are expressed in terms of relations and rules. It's based on a formal system called Horn clauses, which consist of facts and rules. Here's a brief overview:

1. Facts: Facts are statements about relationships between entities. They are represented as predicates.

*Example:*

human(socrates).

1. Rules: Rules define relationships based on conditions. They consist of a head and a body.

*Example:*

mortal(X) :- human(X).

1. Queries: In Prolog, you can ask queries to the knowledge base to retrieve information or verify facts.

*Example:*

?- mortal(socrates).

**IDEs for Prolog Programming:**

There are several IDEs (Integrated Development Environments) available for Prolog programming. Some popular ones include:

* SWI-Prolog: It's a comprehensive Prolog environment with a graphical debugger and IDE-like features. It's available for multiple platforms.
* GNU Prolog: This is a free Prolog compiler with a command-line interface. It provides a basic environment for Prolog development.
* SICStus Prolog: It's a commercial Prolog development system with a comprehensive IDE and advanced features for debugging and optimization.

**11b: Implementing tasks in Prolog**

Medical Diagnosis of Common Cold and Flu Using Symptom Inputs: symptom(fever).

symptom(cough). symptom(sore\_throat). symptom(runny\_nose). symptom(headache). symptom(muscle\_aches). symptom(fatigue).

diagnosis(cold) :symptom(fever), symptom(cough), symptom(runny\_nose), not(symptom(headache)), not(symptom(muscle\_aches)), not(symptom(sore\_throat)), not(symptom(fatigue)).

diagnosis(flu) :symptom(fever), symptom(cough), symptom(runny\_nose), symptom(headache), symptom(muscle\_aches), symptom(fatigue), not(symptom(sore\_throat)).

**Demonstrating Lists in Prolog:**

% Predicate to check if X is a member of the list.

member(X, [X|\_]).

member(X, [\_|T]) :member(X, T).

% Predicate to append two lists.

append([], L, L).

append([H|T], L, [H|R]) :append(T, L, R).

**PRACTICAL 12**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Semester:** | VI | **Division:** | 6 |
| **Roll No.:** | 21BCP359 | **Date:** | 24-04-24 | **Batch:** | G11 |
| **Aim:** | WAP to design Tic Tac Toe games from O (Opponent) and X (Player) by using minimax algorithm. | | | | |

**Program**

import sys

# Function to print the Tic Tac Toe board

def print\_board(board):

    for row in board:

        print(" | ".join(row))

        print(" - " \* 3)

# Function to check if the board is full

def is\_board\_full(board):

    for row in board:

        for cell in row:

            if cell == " ":

                return False

    return True

# Function to check if a player has won

def check\_winner(board, player):

    # Check rows

    for row in board:

        if all(cell == player for cell in row):

            return True

    # Check columns

    for col in range(3):

        if all(board[row][col] == player for row in range(3)):

            return True

    # Check diagonals

    if all(board[i][i] == player for i in range(3)) or all(

        board[i][2 - i] == player for i in range(3)

    ):

        return True

    return False

# Function to evaluate the current state of the board

def evaluate(board):

    if check\_winner(board, "O"):

        return 1

    elif check\_winner(board, "X"):

        return -1

    elif is\_board\_full(board):

        return 0

    else:

        return None

# Minimax algorithm implementation

def minimax(board, depth, is\_maximizing):

    score = evaluate(board)

    if score is not None:

        return score

    if is\_maximizing:

        best\_score = -sys.maxsize

        for i in range(3):

            for j in range(3):

                if board[i][j] == " ":

                    board[i][j] = "O"

                    score = minimax(board, depth + 1, False)

                    board[i][j] = " "

                    best\_score = max(best\_score, score)

        return best\_score

    else:

        best\_score = sys.maxsize

        for i in range(3):

            for j in range(3):

                if board[i][j] == " ":

                    board[i][j] = "X"

                    score = minimax(board, depth + 1, True)

                    board[i][j] = " "

                    best\_score = min(best\_score, score)

        return best\_score

# Function to find the best move for the opponent using Minimax

def find\_best\_move(board):

    best\_score = -sys.maxsize

    best\_move = None

    for i in range(3):

        for j in range(3):

            if board[i][j] == " ":

                board[i][j] = "O"

                score = minimax(board, 0, False)

                board[i][j] = " "

                if score > best\_score:

                    best\_score = score

                    best\_move = (i, j)

    return best\_move

# Main function to play the game

def play\_game():

    board = [[" " for \_ in range(3)] for \_ in range(3)]

    print("Welcome to Tic Tac Toe!")

    print\_board(board)

    while True:

        # Player's move

        row, col = map(int, input("Enter your move (row col): ").split())

        if board[row][col] != " ":

            print("Invalid move. Try again.")

            continue

        board[row][col] = "X"

        print\_board(board)

        if check\_winner(board, "X"):

            print("Congratulations! You win!")

            break

        if is\_board\_full(board):

            print("It's a draw!")

            break

        # Opponent's move

        print("Opponent is thinking...")

        opponent\_row, opponent\_col = find\_best\_move(board)

        board[opponent\_row][opponent\_col] = "O"

        print\_board(board)

        if check\_winner(board, "O"):

            print("Sorry, you lose!")

            break

        if is\_board\_full(board):

            print("It's a draw!")

            break

if \_\_name\_\_ == "\_\_main\_\_":

    play\_game()

**Output**

