**Parallel Implementation of CNN for Visual Search and Product Recommendation**

**Authors:**

Jen Jose Jeeson - 2023BCS0017

Jefin Francis - 2023BCS0053

**Dr. John Paul Martin**

Department of Computer Science, Indian Institute of Information Technology, Kottayam

**Date of Submission:** [10/10/2025]

**Abstract:**

Convolutional Neural Networks (CNNs) is a highly effective deep learning algorithm for visual understanding tasks such as visual search and product recommendation in e commerce, as they can extract features and distinguish unique object characteristics from images. However, CNNs are computationally expensive due to complex operations and the large volume of data, making their sequential execution on CPUs too expensive for real world application.

This project addresses the computational bottleneck by investigating parallel computing methods within the CNN architecture to enhance inference speed and training time. The work specifically focuses on designing and implementing an Openmp based parallel CNN model in C++.

Our aim was to identify parallelizable sections, optimize the computation using Openmp, and then compare the performance and error rate against the serial implementation across different thread counts.

The results that was obtained showed that parallel implementation reduced execution time compared to sequential processing ,ie from 50 seconds to 35 seconds. The best performance in terms of execution time and error rate was achieved while using 4 threads. On 8 threads,the performance decreased due to communication overhead. This project shows the advantage of Openmp based parallelization for accelerating CNN training and testing for visual search and product recommendation.

**1. Introduction**

Convolutional Neural Network is a deep learning algorithm applied to problems like image classification and visual search. Their architecture allows them to get key features as well as learn hierarchially, which makes them a good solution for problems that depend on visual similarity like product recommendation system.

But CNNs are computationally expensive because of the large amount of data and complex operations required for training and inference. It is too costly computationally for serial execution on CPUs to process millions of image features and execute high dimensional convolutions. We can solve this issue by introducing parallelism methods to CNN architecture Parallel computing methods can be added to CNN architectures to solve this. By breaking down computations across many threads we can reduce training time by much and enhance inference speed.

Our work finds how parallelization improves performance and error percentage.We look to implement appropriate parallelization techniques for various layers of CNN, optimize the computations, and analyze their effects on efficiency, in terms of time, and recommendation accuracy, in terms of error %.

**2. Literature Survey:**

There are a few works done on CNNs which have proven the effectiveness of CNNs for image classifying and recommendation. Usual implementations are based on sequential CPU execution, which can be a bottleneck for large data. GPUs are used for this purpose usually, but Openmp offers a CPU based parallelization method.

Some of the works have shown openmps capability for improving the performance on computationally intensive loops, matrix multiplication, and neural network layers. But synchronization overhead can result in poor results, something we investigated and discovered in our research especially when the number of threads are increased beyond a certain level.

**3. Problem Statement and Objectives**

**3.1 Problem Definition**  
CNNs are computationally expensive, especially for large scale image data in e commerce visual search and recommendation systems. The sequential execution of convolution then activation, and backpropagation method on CPU result in really long execution times.

**3.2 Objectives:**

* Implement a serial CNN model in C++.
* Identify parallelizable sections within the CNN operations.
* Design an Openmp-based implementation that improves computation time.
* Compare performance, error rate, and scalability across thread counts.

### 

### 

### 

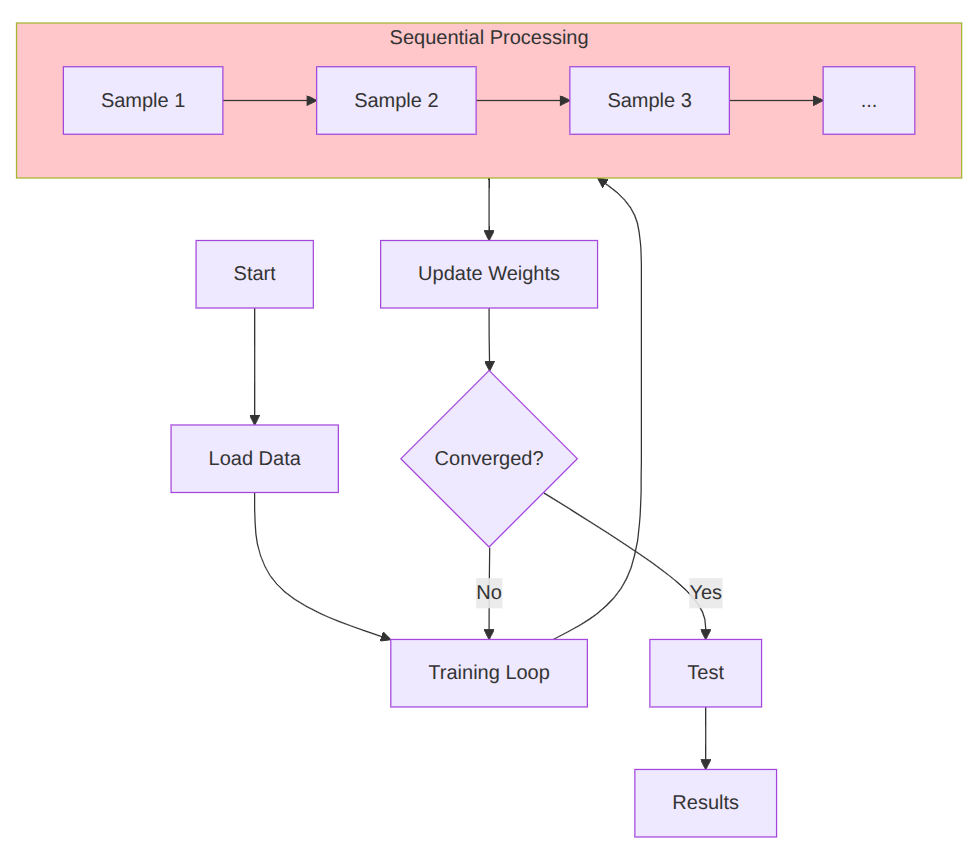
### 

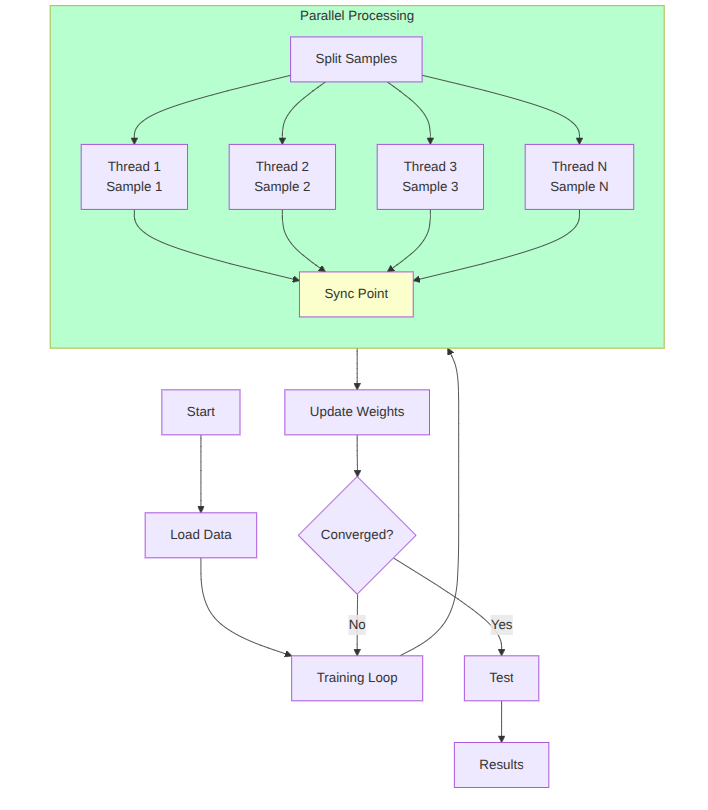
### 

### 

### 

### **4. Methodology and System Architecture**





**4.1 Serial Algorithm**

for (int i = 0; i < train\_cnt; ++i) {

forward\_pass(train\_set[i].data);

l\_f.bp\_clear();

l\_s1.bp\_clear();

l\_c1.bp\_clear();

makeError(l\_f.d\_preact, l\_f.output, train\_set[i].label, 10);

tmp\_err = vectorNorm(l\_f.d\_preact, 10);

err += tmp\_err;

back\_pass();

}

**4.2 Parallel Algorithm (Openmp)**

#pragma omp parallel for private(tmp\_err) reduction(+:err)

for (int i = 0; i < train\_cnt; ++i) {

forward\_pass(train\_set[i].data);

l\_f.bp\_clear();

l\_s1.bp\_clear();

l\_c1.bp\_clear();

makeError(l\_f.d\_preact, l\_f.output, train\_set[i].label, 10);

tmp\_err = vectorNorm(l\_f.d\_preact, 10);

#pragma omp atomic

err += tmp\_err;

back\_pass();

}

Layer Level Parallelization:

#pragma omp parallel for collapse(2)

for (int m = 0; m < 6; ++m) {

for (int x = 0; x < 24; ++x) {

for (int y = 0; y < 24; ++y) {

float sum = 0.0f;

for (int i = 0; i < 5; ++i) {

for (int j = 0; j < 5; ++j) {

sum += input[x + i][y + j] \* weight[m][i][j];

}

}

preact[m][x][y] = sum + bias[m];

}

}

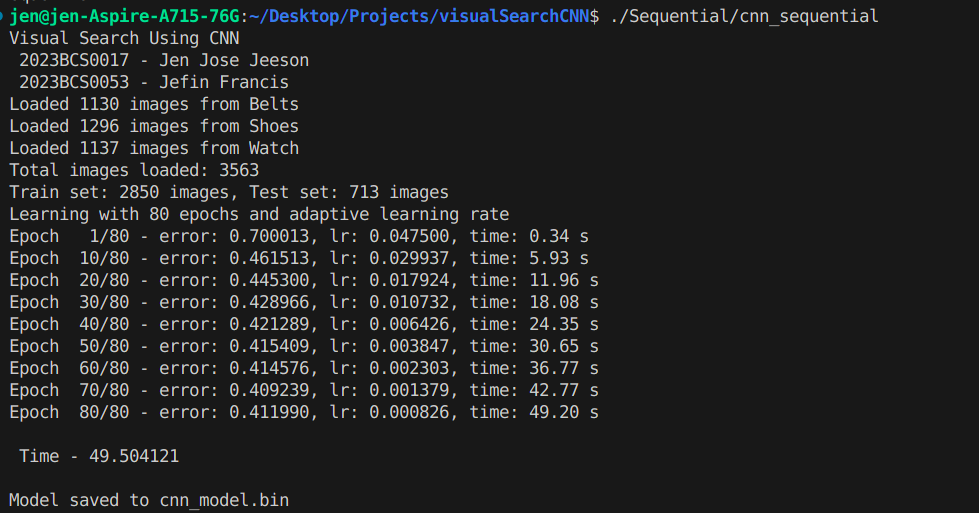
}

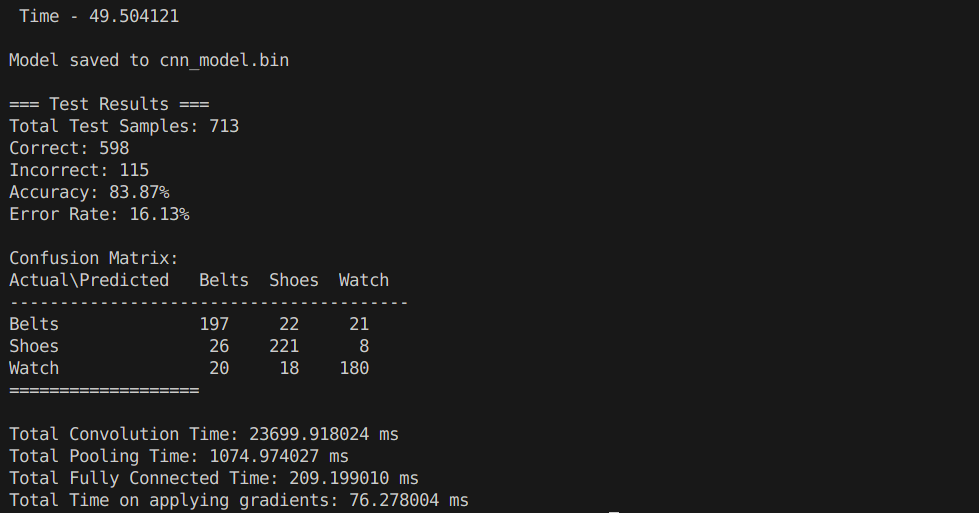
**4.3 Implementation Details**

* Programming Language: C++
* Platform: Linux
* Compiler: G++ (-fOpenmp)
* Hardware Configuration: Intel i5 13th Gen, 10 cores,12 threads, 16 GB RAM

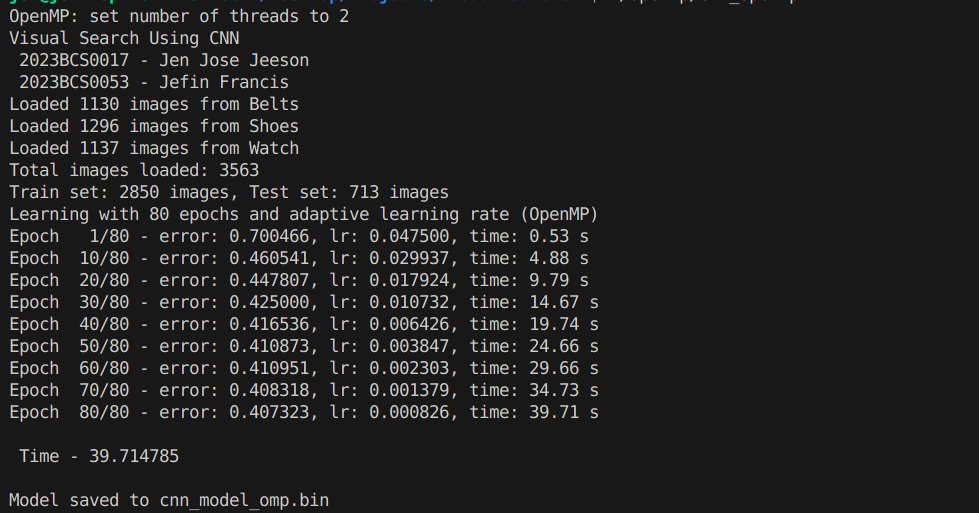
**5. Results and Analysis**

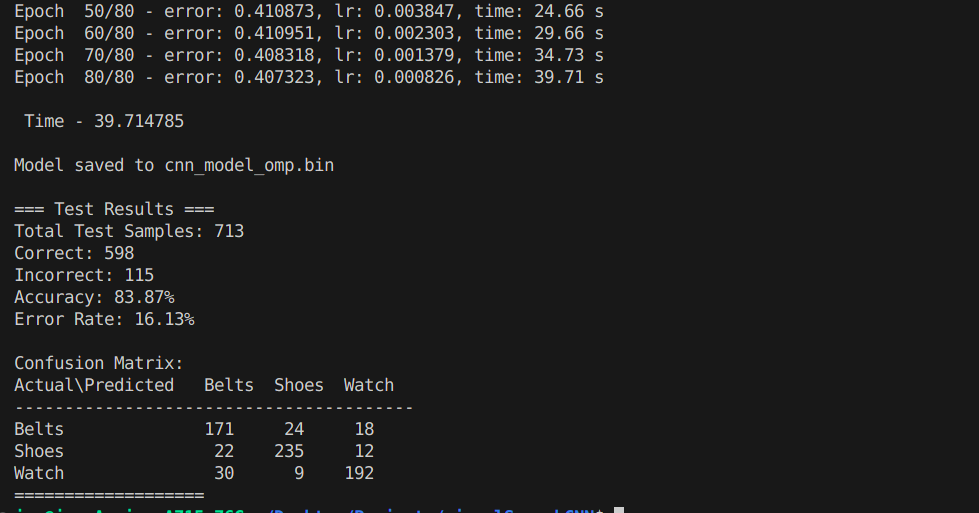
**Sequential:**



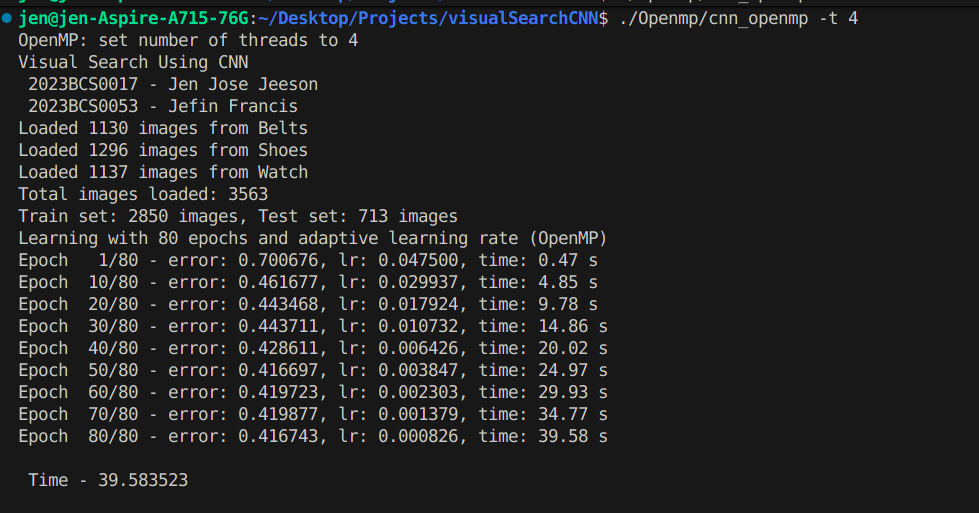


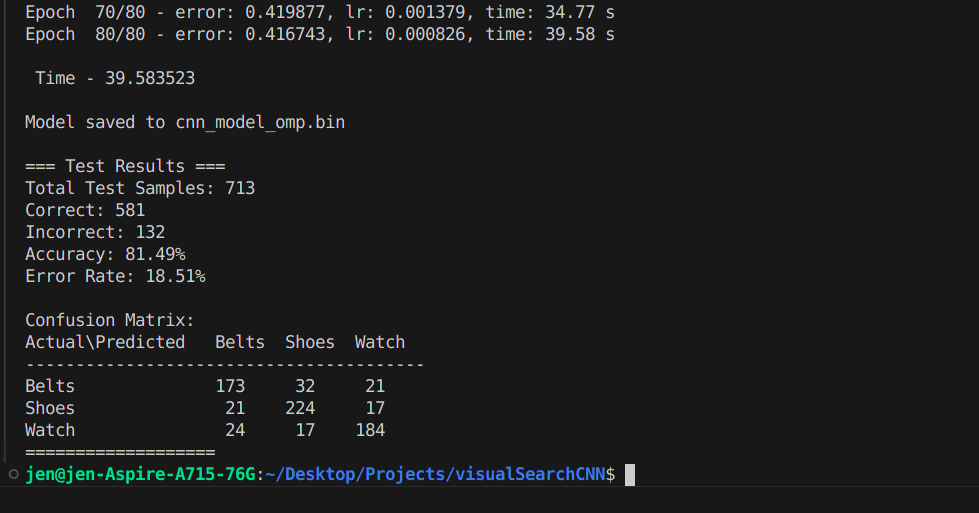
**Openmp on 2 threads:**



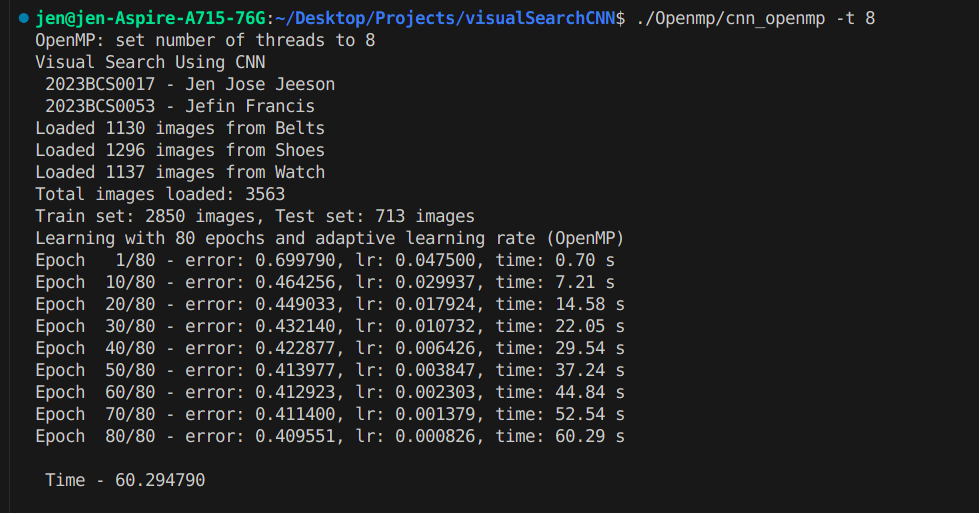


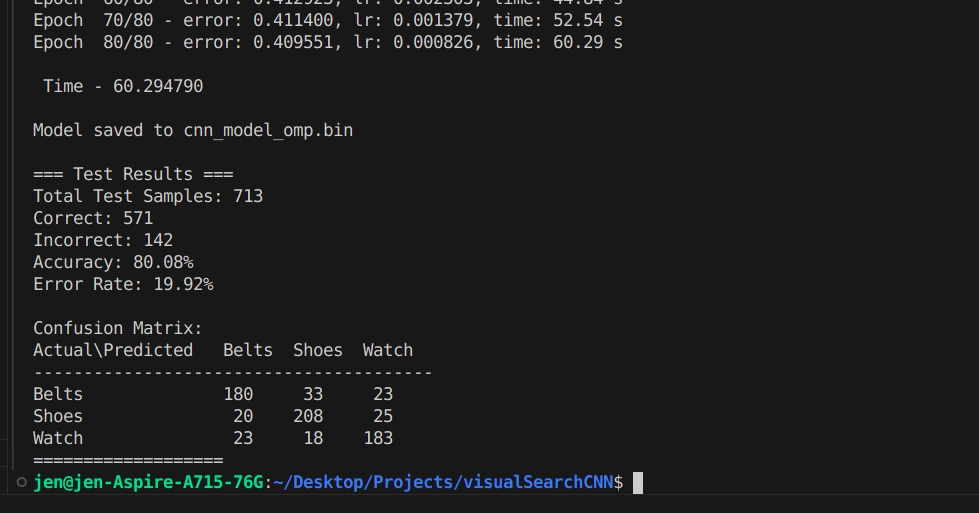
**Openmp on 4 threads:**





**Openmp on 8 threads:**





| Num of threads | Exec Time (in sec) | Error(%) |
| --- | --- | --- |
| 1 | 49.504121 | 16.13 |
| 2 | 39.714785 | 16.13 |
| 4 | 39.583523 | 18.51 |
| 8 | 60.294790 | 19.92 |

**6. Discussion and Observations:**

We found out that we get the best time and error % when number of threads is set to 4. But when number of threads is set to 8, the output becomes worse than sequential execution. The parallel implementation has a linear effect on increasing the error rate. Increasing the number of threads beyond a certain limit can reduce the overall performance and error reduction of the algorithm due to the different overhead.

**7. Conclusion and Future Scope:**

Our project, Parallel Implementation of CNN for Visual Search and Product Recommendation, shows the benefits of using Openmp based parallelization for training and testing a Convolutional Neural Network. This approach can accelerate the process of training.

In the future, this project can extend to:

* Implementation of multi node distributed training using MPI
* Use CUDA for GPU based acceleration in Nvidia GPU’s
* Apply this approach to e-commerce real time platforms and websites.
* Integration of this approach into advanced frameworks like Tensorflow and Pytorch.

**8. References:**

[Parallel Deep Convolutional Neural Network Training by Exploiting the Overlapping of Computation and Communication](http://cucis.ece.northwestern.edu/publications/pdf/LJA17.pdf)

**Appendix**

**Serial Code:**

#include "image\_loader.h"

#include "layer.h"

#include <cstdio>

#include <ctime>

#include <vector>

#include <cmath>

#include <memory>

#include <time.h>

#include <algorithm>

#include <random>

#include <cstring>

#include <thread>

#include <chrono>

double total\_convolution\_time = 0, total\_pooling\_time = 0, total\_fully\_connected\_time = 0,total\_gradient\_time=0;

static image\_data \*train\_set, \*test\_set;

static unsigned int train\_cnt, test\_cnt;

// Define layers of CNN (3 output classes: Belts, Shoes, Watch)

static Layer l\_input(0, 0, 28\*28);

static Layer l\_c1(5\*5, 6, 24\*24\*6);

static Layer l\_s1(4\*4, 1, 6\*6\*6);

static Layer l\_f(6\*6\*6, 3, 3);

static void learn();

static unsigned int classify(double data[28][28]);

static void test();

static double forward\_pass(double data[28][28]);

static double back\_pass();

static void save\_model(const char\* filename);

static bool load\_model(const char\* filename);

static void test\_single\_image(double data[28][28]);

float vectorNorm(float\* vec, int n) {

float sum = 0.0f;

for (int i = 0; i < n; ++i) {

sum += vec[i] \* vec[i];

}

return sqrt(sum);

}

// Simple data augmentation: add random noise

void augment\_image(double original[28][28], double augmented[28][28], float noise\_level = 0.05f) {

for (int i = 0; i < 28; ++i) {

for (int j = 0; j < 28; ++j) {

// Add small random noise

float noise = (static\_cast<float>(rand()) / RAND\_MAX - 0.5f) \* noise\_level;

augmented[i][j] = original[i][j] + noise;

// Clamp to [0, 1]

if (augmented[i][j] < 0.0) augmented[i][j] = 0.0;

if (augmented[i][j] > 1.0) augmented[i][j] = 1.0;

}

}

}

// Horizontal flip augmentation

void flip\_horizontal(double original[28][28], double flipped[28][28]) {

for (int i = 0; i < 28; ++i) {

for (int j = 0; j < 28; ++j) {

flipped[i][27 - j] = original[i][j];

}

}

}

static inline void loaddata()

{

image\_data \*all\_data;

unsigned int total\_count;

// Load all images from the four categories

if (load\_custom\_dataset(&all\_data, &total\_count, "data") != 0) {

fprintf(stderr, "Failed to load dataset\n");

exit(1);

}

// Split into train and test sets (80/20 split)

split\_dataset(all\_data, total\_count, &train\_set, &train\_cnt, &test\_set, &test\_cnt);

free(all\_data);

}

int main(int argc, const char \*\*argv) {

const char\* model\_file = "cnn\_model.bin";

bool skip\_training = false;

double custom\_image[28][28];

bool test\_custom = false;

bool run\_full\_test = true;

srand(time(NULL));

// Parse command line arguments first

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

if (strcmp(argv[i], "--load") == 0 || strcmp(argv[i], "-l") == 0) {

skip\_training = true;

} else if (strcmp(argv[i], "--model") == 0 || strcmp(argv[i], "-m") == 0) {

if (i + 1 < argc) {

model\_file = argv[++i];

}

} else if (strcmp(argv[i], "--test-image") == 0 || strcmp(argv[i], "-i") == 0) {

if (i + 1 < argc) {

const char\* image\_path = argv[++i];

if (load\_single\_image(image\_path, custom\_image) == 0) {

test\_custom = true;

fprintf(stdout, "\n=== Testing Custom Image ===\n");

fprintf(stdout, "Image: %s\n", image\_path);

} else {

fprintf(stderr, "Failed to load image: %s\n", image\_path);

return 1;

}

}

} else if (strcmp(argv[i], "--no-test") == 0) {

run\_full\_test = false;

} else if (strcmp(argv[i], "--help") == 0 || strcmp(argv[i], "-h") == 0) {

printf("Usage: %s [OPTIONS]\n", argv[0]);

printf("Options:\n");

printf(" --load, -l Load pre-trained model instead of training\n");

printf(" --model, -m <file> Specify model file (default: cnn\_model.bin)\n");

printf(" --test-image, -i <file> Test a single custom image\n");

printf(" --no-test Skip validation dataset testing\n");

printf(" --help, -h Show this help message\n");

printf("\nExample: ./cnn\_sequential --load -i myimage.jpg --no-test\n");

return 0;

}

}

// If just testing a custom image with loaded model, skip dataset loading

if (skip\_training && test\_custom && !run\_full\_test) {

if (load\_model(model\_file)) {

fprintf(stdout, "Model loaded from %s\n\n", model\_file);

test\_single\_image(custom\_image);

return 0;

} else {

fprintf(stderr, "Failed to load model from %s\n", model\_file);

return 1;

}

}

fprintf(stdout ,"Visual Search Using CNN\n 2023BCS0017 - Jen Jose Jeeson\n 2023BCS0053 - Jefin Francis\n");

// Load dataset only if we need to train or run full test

loaddata();

if (test\_custom) {

test\_single\_image(custom\_image);

}

if (run\_full\_test) {

test();

}

printf("\nTotal Convolution Time: %f ms\n", total\_convolution\_time);

printf("Total Pooling Time: %f ms\n", total\_pooling\_time);

printf("Total Fully Connected Time: %f ms\n", total\_fully\_connected\_time);

printf("Total Time on applying gradients: %f ms\n", total\_gradient\_time);

return 0;

}

static double forward\_pass(double data[28][28]) {

float input[28][28];

for (int i = 0; i < 28; ++i) {

for (int j = 0; j < 28; ++j) {

input[i][j] = data[i][j];

}

}

l\_input.clear();

l\_c1.clear();

l\_s1.clear();

l\_f.clear();

float milliseconds=0;

clock\_t start, end;

clock\_t start\_1, end\_1;

start\_1=clock();

l\_input.setOutput((float \*)input);

// forward pass Convolution Layer

start = clock();

fp\_c1((float (\*)[28])l\_input.output, (float (\*)[24][24])l\_c1.preact, (float (\*)[5][5])l\_c1.weight,l\_c1.bias);

apply\_step\_function(l\_c1.preact, l\_c1.output, l\_c1.O);

end = clock();

milliseconds = 1000.0 \* (end - start) / CLOCKS\_PER\_SEC;

total\_convolution\_time += milliseconds;

// forward pass pooling Layer

start = clock();

fp\_s1((float (\*)[24][24])l\_c1.output, (float (\*)[6][6])l\_s1.preact, (float (\*)[4][4])l\_s1.weight,l\_s1.bias);

apply\_step\_function(l\_s1.preact, l\_s1.output, l\_s1.O);

end = clock();

milliseconds = 1000.0 \* (end - start) / CLOCKS\_PER\_SEC;

total\_pooling\_time += milliseconds;

// forward pass Fully Connected Layer

start = clock();

fp\_preact\_f((float (\*)[6][6])l\_s1.output, l\_f.preact, l\_f.weight, l\_f.N);

fp\_bias\_f(l\_f.preact, l\_f.bias, l\_f.N);

apply\_step\_function(l\_f.preact, l\_f.output, l\_f.O);

end = clock();

milliseconds = 1000.0 \* (end - start) / CLOCKS\_PER\_SEC;

total\_fully\_connected\_time += milliseconds;

end\_1= clock();

return ((double) (end\_1 - start\_1)) / CLOCKS\_PER\_SEC;

}

static double back\_pass() {

clock\_t start,end;

clock\_t start\_1,end\_1;

start\_1=clock();

float milliseconds=0;

start = clock();

bp\_weight\_f(l\_f.d\_weight, l\_f.d\_preact, (float (\*)[6][6])l\_s1.output, l\_f.N);

bp\_bias\_f(l\_f.bias, l\_f.d\_preact, l\_f.N);

end = clock();

milliseconds = 1000.0 \* (end - start) / CLOCKS\_PER\_SEC;

total\_fully\_connected\_time += milliseconds;

start = clock();

bp\_output\_s1((float (\*)[6][6])l\_s1.d\_output, l\_f.weight, l\_f.d\_preact, l\_f.N);

bp\_preact\_s1((float (\*)[6][6])l\_s1.d\_preact, (float (\*)[6][6])l\_s1.d\_output, (float (\*)[6][6])l\_s1.preact);

bp\_weight\_s1((float (\*)[4][4])l\_s1.d\_weight, (float (\*)[6][6])l\_s1.d\_preact, (float (\*)[24][24])l\_c1.output);

bp\_bias\_s1(l\_s1.bias, (float (\*)[6][6])l\_s1.d\_preact);

end = clock();

milliseconds = 1000.0 \* (end - start) / CLOCKS\_PER\_SEC;

total\_pooling\_time += milliseconds;

start = clock();

bp\_output\_c1((float (\*)[24][24])l\_c1.d\_output, (float (\*)[4][4])l\_s1.weight, (float (\*)[6][6])l\_s1.d\_preact);

bp\_preact\_c1((float (\*)[24][24])l\_c1.d\_preact, (float (\*)[24][24])l\_c1.d\_output, (float (\*)[24][24])l\_c1.preact);

bp\_weight\_c1((float (\*)[5][5])l\_c1.d\_weight, (float (\*)[24][24])l\_c1.d\_preact, (float (\*)[28])l\_input.output);

bp\_bias\_c1(l\_c1.bias, (float (\*)[24][24])l\_c1.d\_preact);

end = clock();

milliseconds = 1000.0 \* (end - start) / CLOCKS\_PER\_SEC;

total\_convolution\_time += milliseconds;

start = clock();

apply\_grad(l\_f.weight, l\_f.d\_weight, l\_f.M \* l\_f.N);

apply\_grad(l\_s1.weight, l\_s1.d\_weight, l\_s1.M \* l\_s1.N);

apply\_grad(l\_c1.weight, l\_c1.d\_weight, l\_c1.M \* l\_c1.N);

end = clock();

milliseconds = 1000.0 \* (end - start) / CLOCKS\_PER\_SEC;

total\_gradient\_time += milliseconds;

end\_1= clock();

return ((double) (end\_1- start\_1)) / CLOCKS\_PER\_SEC;

}

static void learn() {

float err;

int total\_epochs = 80;

int iter = total\_epochs;

int current\_epoch = 0;

double time\_taken = 0.0;

fprintf(stdout ,"Learning with %d epochs and adaptive learning rate\n", total\_epochs);

while (iter < 0 || iter-- > 0) {

current\_epoch++;

// Update learning rate with decay

update\_learning\_rate(current\_epoch, total\_epochs);

err = 0.0f;

// Shuffle training indices for randomization

std::vector<int> indices(train\_cnt);

for(int i = 0; i < train\_cnt; ++i) indices[i] = i;

std::shuffle(indices.begin(), indices.end(), std::default\_random\_engine(time(NULL) + current\_epoch));

for (int idx : indices) {

float tmp\_err;

// Randomly augment data (50% chance)

double augmented\_data[28][28];

if (rand() % 2 == 0 && current\_epoch > 10) { // Start augmentation after 10 epochs

if (rand() % 2 == 0) {

augment\_image(train\_set[idx].data, augmented\_data, 0.05f);

} else {

flip\_horizontal(train\_set[idx].data, augmented\_data);

}

time\_taken += forward\_pass(augmented\_data);

} else {

time\_taken += forward\_pass(train\_set[idx].data);

}

l\_f.bp\_clear();

l\_s1.bp\_clear();

l\_c1.bp\_clear();

// Euclid distance of train\_set[idx]

makeError(l\_f.d\_preact, l\_f.output, train\_set[idx].label, 3);

tmp\_err = vectorNorm(l\_f.d\_preact, 3);

err += tmp\_err;

time\_taken += back\_pass();

}

err /= train\_cnt;

// Print progress every 10 epochs or if error is very low

if (current\_epoch % 10 == 0 || current\_epoch == 1 || err < 0.15) {

extern float dt; // Access current learning rate

fprintf(stdout, "Epoch %3d/%d - error: %.6f, lr: %.6f, time: %.2lf s\n",

current\_epoch, total\_epochs, err, dt, time\_taken);

}

if (err < threshold) {

fprintf(stdout, "Training complete, error less than threshold\n\n");

break;

}

}

fprintf(stdout, "\n Time - %lf\n", time\_taken);

}

static unsigned int classify(double data[28][28]) {

float res[3];

forward\_pass(data);

unsigned int max = 0;

for (int i = 0; i < 3; i++) {

res[i] = l\_f.output[i];

}

for (int i = 1; i < 3; ++i) {

if (res[max] < res[i]) {

max = i;

}

}

return max;

}

static void test()

{

int error = 0;

const char\* class\_names[] = {"Belts", "Shoes", "Watch"};

int confusion\_matrix[3][3] = {0};

for (int i = 0; i < test\_cnt; ++i) {

unsigned int predicted = classify(test\_set[i].data);

unsigned int actual = test\_set[i].label;

confusion\_matrix[actual][predicted]++;

if (predicted != actual) {

++error;

}

}

fprintf(stdout, "\n=== Test Results ===\n");

fprintf(stdout, "Total Test Samples: %d\n", test\_cnt);

fprintf(stdout, "Correct: %d\n", test\_cnt - error);

fprintf(stdout, "Incorrect: %d\n", error);

fprintf(stdout, "Accuracy: %.2lf%%\n",

(1.0 - double(error) / double(test\_cnt)) \* 100.0);

fprintf(stdout, "Error Rate: %.2lf%%\n\n",

double(error) / double(test\_cnt) \* 100.0);

fprintf(stdout, "Confusion Matrix:\n");

fprintf(stdout, "Actual\\Predicted Belts Shoes Watch\n");

fprintf(stdout, "----------------------------------------\n");

for (int i = 0; i < 3; i++) {

fprintf(stdout, "%-15s", class\_names[i]);

for (int j = 0; j < 3; j++) {

fprintf(stdout, "%7d", confusion\_matrix[i][j]);

}

fprintf(stdout, "\n");

}

fprintf(stdout, "===================\n");

}

static void test\_single\_image(double data[28][28]) {

unsigned int prediction = classify(data);

const char\* class\_names[] = {"Belts", "Shoes", "Watch"};

fprintf(stdout, "\n=== Prediction Results ===\n");

fprintf(stdout, "Predicted class: %s (label %d)\n", class\_names[prediction], prediction);

fprintf(stdout, "\nConfidence scores:\n");

for (int i = 0; i < 3; i++) {

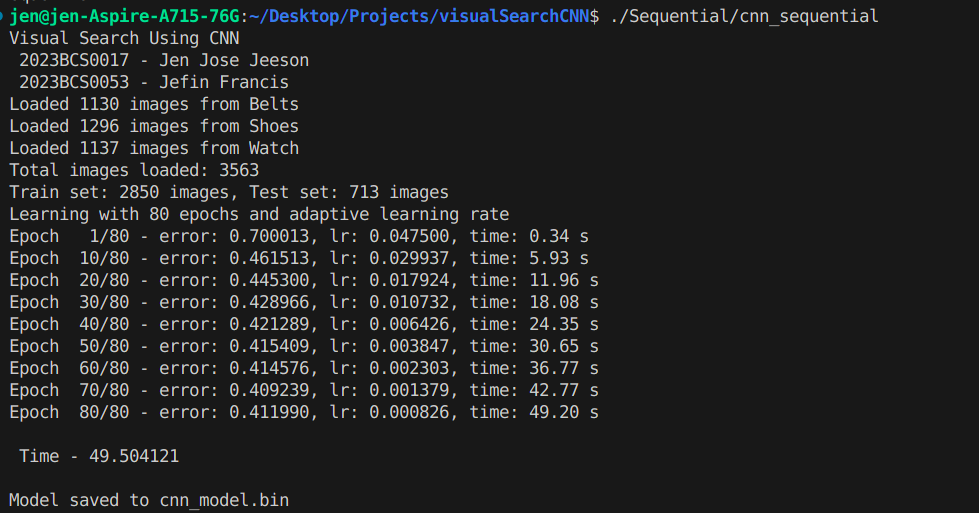
fprintf(stdout, " %s: %.4f\n", class\_names[i], l\_f.output[i]);

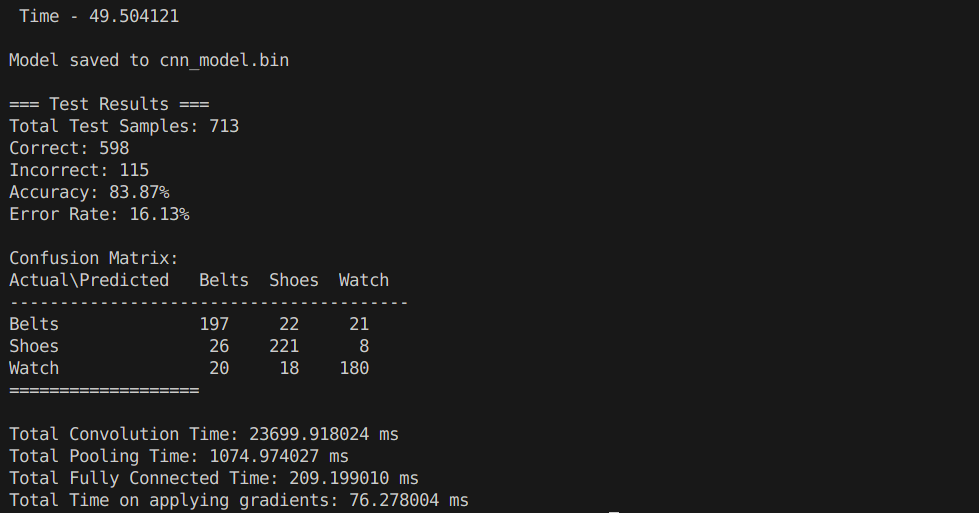
}

fprintf(stdout, "========================\n\n");

}

**Serial Output:**





**Parallel Code:**

#include "image\_loader.h"

#include "layer.h"

#include <cstdio>

#include <ctime>

#include <vector>

#include <cmath>

#include <memory>

#include <time.h>

#include <cstring>

#include <cstdlib>

#include <algorithm>

#include <random>

#include <omp.h>

static image\_data \*train\_set, \*test\_set;

static unsigned int train\_cnt, test\_cnt;

// Define layers of CNN (3 output classes: Belts, Shoes, Watch)

static Layer l\_input(0, 0, 28\*28);

static Layer l\_c1(5\*5, 6, 24\*24\*6);

static Layer l\_s1(4\*4, 1, 6\*6\*6);

static Layer l\_f(6\*6\*6, 3, 3);

static void learn();

static unsigned int classify(double data[28][28]);

static void test();

static double forward\_pass(double data[28][28]);

static double back\_pass();

static void save\_model(const char\* filename);

static bool load\_model(const char\* filename);

static void test\_single\_image(double data[28][28]);

float vectorNorm(float\* vec, int n) {

float sum = 0.0f;

for (int i = 0; i < n; ++i) {

sum += vec[i] \* vec[i];

}

return sqrt(sum);

}

// Simple data augmentation: add random noise

void augment\_image(double original[28][28], double augmented[28][28], float noise\_level = 0.05f) {

for (int i = 0; i < 28; ++i) {

for (int j = 0; j < 28; ++j) {

// Add small random noise

float noise = (static\_cast<float>(rand()) / RAND\_MAX - 0.5f) \* noise\_level;

augmented[i][j] = original[i][j] + noise;

// Clamp to [0, 1]

if (augmented[i][j] < 0.0) augmented[i][j] = 0.0;

if (augmented[i][j] > 1.0) augmented[i][j] = 1.0;

}

}

}

// Horizontal flip augmentation

void flip\_horizontal(double original[28][28], double flipped[28][28]) {

for (int i = 0; i < 28; ++i) {

for (int j = 0; j < 28; ++j) {

flipped[i][27 - j] = original[i][j];

}

}

}

static inline void loaddata()

{

image\_data \*all\_data;

unsigned int total\_count;

// Load all images from the four categories

if (load\_custom\_dataset(&all\_data, &total\_count, "data") != 0) {

fprintf(stderr, "Failed to load dataset\n");

exit(1);

}

// Split into train and test sets (80/20 split)

split\_dataset(all\_data, total\_count, &train\_set, &train\_cnt, &test\_set, &test\_cnt);

free(all\_data);

}

int main(int argc, const char \*\*argv) {

int num\_threads = 0;

const char\* model\_file = "cnn\_model\_omp.bin";

bool skip\_training = false;

const char\* test\_image\_path = nullptr;

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

if (strcmp(argv[i], "-t") == 0 || strcmp(argv[i], "--threads") == 0) {

if (i + 1 < argc) {

num\_threads = atoi(argv[++i]);

}

} else if (strcmp(argv[i], "--load") == 0 || strcmp(argv[i], "-l") == 0) {

skip\_training = true;

} else if (strcmp(argv[i], "--model") == 0 || strcmp(argv[i], "-m") == 0) {

if (i + 1 < argc) {

model\_file = argv[++i];

}

} else if (strcmp(argv[i], "--test-image") == 0 || strcmp(argv[i], "-i") == 0) {

if (i + 1 < argc) {

test\_image\_path = argv[++i];

}

} else if (strcmp(argv[i], "-h") == 0 || strcmp(argv[i], "--help") == 0) {

fprintf(stdout, "Usage: %s [OPTIONS]\n", argv[0]);

fprintf(stdout, "Options:\n");

fprintf(stdout, " -t, --threads <N> Set number of OpenMP threads\n");

fprintf(stdout, " --load, -l Load pre-trained model instead of training\n");

fprintf(stdout, " --model, -m <file> Specify model file (default: cnn\_model\_omp.bin)\n");

fprintf(stdout, " --test-image, -i <file> Test a custom image and show prediction\n");

fprintf(stdout, " --help, -h Show this help message\n");

fprintf(stdout, "\nExamples:\n");

fprintf(stdout, " %s -t 8 # Train with 8 threads\n", argv[0]);

fprintf(stdout, " %s --load -t 4 # Load model, test with 4 threads\n", argv[0]);

fprintf(stdout, " %s --load --test-image shoe.jpg # Test custom image\n", argv[0]);

fprintf(stdout, " %s -t 8 --test-image data/Shoes/s1.jpg # Train and test custom image\n", argv[0]);

return 0;

} else if (argv[i][0] >= '0' && argv[i][0] <= '9') {

/\* allow a bare numeric positional argument \*/

num\_threads = atoi(argv[i]);

}

}

if (num\_threads > 0) {

omp\_set\_num\_threads(num\_threads);

fprintf(stdout, "OpenMP: set number of threads to %d\n", num\_threads);

} else {

fprintf(stdout, "OpenMP: using default number of threads (%d)\n", omp\_get\_max\_threads());

}

srand(time(NULL));

// Check if we only need to test a custom image with a loaded model

bool only\_test\_image = skip\_training && test\_image\_path != nullptr;

// If just testing custom image, skip dataset loading

if (only\_test\_image) {

if (load\_model(model\_file)) {

fprintf(stdout, "Model loaded from %s\n\n", model\_file);

double custom\_data[28][28];

if (load\_single\_image(test\_image\_path, custom\_data) == 0) {

fprintf(stdout, "=== Testing Custom Image ===\n");

fprintf(stdout, "Image: %s\n", test\_image\_path);

test\_single\_image(custom\_data);

return 0;

} else {

fprintf(stderr, "Failed to load custom image: %s\n", test\_image\_path);

return 1;

}

} else {

fprintf(stderr, "Failed to load model from %s\n", model\_file);

return 1;

}

}

fprintf(stdout ,"Visual Search Using CNN\n 2023BCS0017 - Jen Jose Jeeson\n 2023BCS0053 - Jefin Francis\n");

// Load dataset for training or full testing

loaddata();

// Test custom image if provided

if (test\_image\_path) {

double custom\_data[28][28];

if (load\_single\_image(test\_image\_path, custom\_data) == 0) {

fprintf(stdout, "\n=== Testing Custom Image ===\n");

fprintf(stdout, "Image: %s\n", test\_image\_path);

test\_single\_image(custom\_data);

} else {

fprintf(stderr, "Failed to load custom image: %s\n", test\_image\_path);

}

}

test();

return 0;

}

static double forward\_pass(double data[28][28]) {

float input[28][28];

for (int i = 0; i < 28; ++i) {

for (int j = 0; j < 28; ++j) {

input[i][j] = data[i][j];

}

}

l\_input.clear();

l\_c1.clear();

l\_s1.clear();

l\_f.clear();

double start\_1 = omp\_get\_wtime();

l\_input.setOutput((float \*)input);

// forward pass Convolution Layer

fp\_c1((float (\*)[28])l\_input.output, (float (\*)[24][24])l\_c1.preact, (float (\*)[5][5])l\_c1.weight,l\_c1.bias);

apply\_step\_function(l\_c1.preact, l\_c1.output, l\_c1.O);

fp\_s1((float (\*)[24][24])l\_c1.output, (float (\*)[6][6])l\_s1.preact, (float (\*)[4][4])l\_s1.weight,l\_s1.bias);

apply\_step\_function(l\_s1.preact, l\_s1.output, l\_s1.O);

// forward pass Fully Connected Layer

fp\_preact\_f((float (\*)[6][6])l\_s1.output, l\_f.preact, l\_f.weight, l\_f.N);

fp\_bias\_f(l\_f.preact, l\_f.bias, l\_f.N);

apply\_step\_function(l\_f.preact, l\_f.output, l\_f.O);

double end\_1 = omp\_get\_wtime();

return end\_1 - start\_1;

}

static double back\_pass() {

double start\_1 = omp\_get\_wtime();

bp\_weight\_f(l\_f.d\_weight, l\_f.d\_preact, (float (\*)[6][6])l\_s1.output, l\_f.N);

bp\_bias\_f(l\_f.bias, l\_f.d\_preact, l\_f.N);

bp\_output\_s1((float (\*)[6][6])l\_s1.d\_output, l\_f.weight, l\_f.d\_preact, l\_f.N);

bp\_preact\_s1((float (\*)[6][6])l\_s1.d\_preact, (float (\*)[6][6])l\_s1.d\_output, (float (\*)[6][6])l\_s1.preact);

bp\_weight\_s1((float (\*)[4][4])l\_s1.d\_weight, (float (\*)[6][6])l\_s1.d\_preact, (float (\*)[24][24])l\_c1.output);

bp\_bias\_s1(l\_s1.bias, (float (\*)[6][6])l\_s1.d\_preact);

bp\_output\_c1((float (\*)[24][24])l\_c1.d\_output, (float (\*)[4][4])l\_s1.weight, (float (\*)[6][6])l\_s1.d\_preact);

bp\_preact\_c1((float (\*)[24][24])l\_c1.d\_preact, (float (\*)[24][24])l\_c1.d\_output, (float (\*)[24][24])l\_c1.preact);

bp\_weight\_c1((float (\*)[5][5])l\_c1.d\_weight, (float (\*)[24][24])l\_c1.d\_preact, (float (\*)[28])l\_input.output);

bp\_bias\_c1(l\_c1.bias, (float (\*)[24][24])l\_c1.d\_preact);

apply\_grad(l\_f.weight, l\_f.d\_weight, l\_f.M \* l\_f.N);

apply\_grad(l\_s1.weight, l\_s1.d\_weight, l\_s1.M \* l\_s1.N);

apply\_grad(l\_c1.weight, l\_c1.d\_weight, l\_c1.M \* l\_c1.N);

double end\_1 = omp\_get\_wtime();

return end\_1 - start\_1;

}

static void learn() {

float err;

int total\_epochs = 80; // Total epochs for high accuracy

int iter = total\_epochs;

int current\_epoch = 0;

double time\_taken = 0.0;

fprintf(stdout ,"Learning with %d epochs and adaptive learning rate (OpenMP)\n", total\_epochs);

while (iter < 0 || iter-- > 0) {

current\_epoch++;

// Update learning rate with decay

update\_learning\_rate(current\_epoch, total\_epochs);

err = 0.0f;

// Shuffle training indices for randomization

std::vector<int> indices(train\_cnt);

for(int i = 0; i < train\_cnt; ++i) indices[i] = i;

std::shuffle(indices.begin(), indices.end(), std::default\_random\_engine(time(NULL) + current\_epoch));

for (int idx : indices) {

float tmp\_err;

// Randomly augment data (50% chance)

double augmented\_data[28][28];

if (rand() % 2 == 0 && current\_epoch > 10) { // Start augmentation after 10 epochs

if (rand() % 2 == 0) {

augment\_image(train\_set[idx].data, augmented\_data, 0.05f);

} else {

flip\_horizontal(train\_set[idx].data, augmented\_data);

}

time\_taken += forward\_pass(augmented\_data);

} else {

time\_taken += forward\_pass(train\_set[idx].data);

}

l\_f.bp\_clear();

l\_s1.bp\_clear();

l\_c1.bp\_clear();

// Euclid distance of train\_set[idx]

makeError(l\_f.d\_preact, l\_f.output, train\_set[idx].label, 3);

tmp\_err = vectorNorm(l\_f.d\_preact, 3);

err += tmp\_err;

time\_taken += back\_pass();

}

err /= train\_cnt;

// Print progress every 10 epochs or if error is very low

if (current\_epoch % 10 == 0 || current\_epoch == 1 || err < 0.15) {

extern float dt; // Access current learning rate

fprintf(stdout, "Epoch %3d/%d - error: %.6f, lr: %.6f, time: %.2lf s\n",

current\_epoch, total\_epochs, err, dt, time\_taken);

}

if (err < threshold) {

fprintf(stdout, "Training complete, error less than threshold\n\n");

break;

}

}

fprintf(stdout, "\n Time - %lf\n", time\_taken);

}

static unsigned int classify(double data[28][28]) {

float res[3];

forward\_pass(data);

unsigned int max = 0;

for (int i = 0; i < 3; i++) {

res[i] = l\_f.output[i];

}

for (int i = 1; i < 3; ++i) {

if (res[max] < res[i]) {

max = i;

}

}

return max;

}

static void test()

{

int error = 0;

const char\* class\_names[] = {"Belts", "Shoes", "Watch"};

int confusion\_matrix[3][3] = {0}; // [actual][predicted]

for (int i = 0; i < test\_cnt; ++i) {

unsigned int predicted = classify(test\_set[i].data);

unsigned int actual = test\_set[i].label;

confusion\_matrix[actual][predicted]++;

if (predicted != actual) {

++error;

}

}

fprintf(stdout, "\n=== Test Results ===\n");

fprintf(stdout, "Total Test Samples: %d\n", test\_cnt);

fprintf(stdout, "Correct: %d\n", test\_cnt - error);

fprintf(stdout, "Incorrect: %d\n", error);

fprintf(stdout, "Accuracy: %.2lf%%\n",

(1.0 - double(error) / double(test\_cnt)) \* 100.0);

fprintf(stdout, "Error Rate: %.2lf%%\n\n",

double(error) / double(test\_cnt) \* 100.0);

fprintf(stdout, "Confusion Matrix:\n");

fprintf(stdout, "Actual\\Predicted Belts Shoes Watch\n");

fprintf(stdout, "----------------------------------------\n");

for (int i = 0; i < 3; i++) {

fprintf(stdout, "%-15s", class\_names[i]);

for (int j = 0; j < 3; j++) {

fprintf(stdout, "%7d", confusion\_matrix[i][j]);

}

fprintf(stdout, "\n");

}

fprintf(stdout, "===================\n");

}

// Test a single custom image

static void test\_single\_image(double data[28][28]) {

unsigned int prediction = classify(data);

const char\* class\_names[] = {"Belts", "Shoes", "Watch"};

fprintf(stdout, "\n=== Prediction Results ===\n");

fprintf(stdout, "Predicted class: %s (label %d)\n", class\_names[prediction], prediction);

fprintf(stdout, "\nConfidence scores:\n");

for (int i = 0; i < 3; i++) {

fprintf(stdout, " %s: %.4f\n", class\_names[i], l\_f.output[i]);

}

fprintf(stdout, "========================\n\n");

}

**Parallel Output:**

