CSC3002 (23 Fall) Assignment 6

Handwritten Digit Recognition using C++ and OpenCV

School of Data Science

Fu Chen, 122090020

December 4, 2023

Introduction

Handwritten digit recognition is a technique that automatically classifies an image to numbers. People may transfer any photo with a number as the main body to workable numbers in computers. In this way, manually typing numbers to computers is no longer needed, which can help improve the efficiency in the areas of accounting, postal services, education, etc.

In this project, the data set MNIST (https://yann.lecun.com/exdb/mnist/) is used to train the model and test the accuracy. Based on the K-nearest neighbors algorithm (KNN), the model is well trained and can predict the number with an accuracy around 96.6%.

Reproducing results

(a) For Windows 10/11 (64-bit) users, an executable file is provided.

1. Find the MNIST folder under the same directory of this report (the folder and this report should both be under the root directory of the zip file).
2. Double click MNIST.exe.

(b) For users working with other systems, a Visual Studio environment is needed.

1. The program is perfectly supported under the environment below. Please try to match the environment as close as possible. Note that the OpenCV library and the compiler should be added to system paths of environment variables.
   - Microsoft Visual Studio Community 2022 (64-bit) Version 17.8.2
   - OpenCV 4.8.1 (Windows release binaries)
   - Compiler: MSVC v142 (Visual Studio 2019 C++ x64/x86 build tools)
2. With a workable environment, open the MNIST folder under the same directory of this report with Microsoft Visual Studio.
3. Choose Debug mode and (x64) platform. Click to run without debugging.

Either way, the MNIST data set (four folders) and 3 test images (sample1/2/3.png) are provided under the same folder for testing, as mentioned in the Program Details section.

Program details

See main.cpp under the MNIST folder.

The KNN algorithm is a supervised clustering algorithm. In this project, the pixels are projected as points in Euclidean space, and the KNN algorithm gives 10 clusters representing each digit. The main function of this program is to train a KNN model (optional) and predict the numbers according to the pictures user provides. The program can be roughly divided into three parts: greeting & guidelines, training & testing (optional), and applicable predictions.

(1) greeting & guidelines

Some introductory message is printed in the first three lines. Then users are asked whether they want to use the pre-trained model. Since other model takes about 4 minutes to train and test on the whole data set, it is time consuming to train the models every time. Therefore, a pre-trained model (with accuracy 96.61% on the test set) is provided in MNIST/model_knn.xml. The object `pre_train` restores the option.

(in main function)

```
// Welcome message and guidelines to the user.
printf("Welcome! This is the assignment 6 of CSC3002 by Chen Fu (122090020).\n");
printf("This program implements a KNN model to recognize handwritten digits.\n");
printf("The MNIST dataset is used to train and test the model.\n\n");
printf("A pre-trained model is provided. You can use it directly or train a new one.\n");
printf("Otherwise you may need about 4 mins to train a new model.\n");
printf("Use the pre-trained model? ([Y]/N) ");
std::string pre_train;
std::getline(std::cin, pre_train);
while(pre_train != "Y" && pre_train != "y" && pre_train != "N" && pre_train != "n" && pre_train != ""){
    printf("Use the pre-trained model? ([Y]/N) ");
    std::getline(std::cin, pre_train);
}
```

(2) training & testing (optional)

If the user choose to use the pre-trained model, then this section is skipped. Otherwise, the data sets are first loaded. Here data sets under the same folder is used, and the `load_data` function is explained later.

(in main function)

```
std::string train_img_path = "train-images-idx3-ubyte/train-images-idx3-ubyte";
std::string train_lab_path = "train-labels-idx1-ubyte/train-labels-idx1-ubyte";
std::string test_img_path = "t10k-images-idx3-ubyte/t10k-images-idx3-ubyte";
std::string test_lab_path = "t10k-labels-idx1-ubyte/t10k-labels-idx1-ubyte";
load_data(train_img_path.c_str(), train_lab_path.c_str(), train_images, train_labels, "train");
load_data(test_img_path.c_str(), test_lab_path.c_str(), test_images, test_labels, "test");
```

Due to the special format of MNIST data set, some functions are designed to read the data. In MNIST, the numbers are restored in big-endian format. When the number contains more than one byte, the byte order should be reversed, as implemented in `_byteswap32` function. Some other functions are used to read numbers with exact bytes.

(global definition)

```
inline uint32_t _byteswap32(uint32_t v){
    return ((v & 0xff000000) >> 24) |
           ((v & 0x00ff0000) >>  8) |
           ((v & 0x0000ff00) <<  8) |
           ((v & 0x000000ff) << 24);
}
inline void _read32(std::ifstream& fin, uint32_t& v){
    fin.read(reinterpret_cast<char*>(&v), sizeof(v));
    v = _byteswap32(v);
}
inline void _read8(std::ifstream& fin, uint8_t& v){
    fin.read(reinterpret_cast<char*>(&v), sizeof(v));
}
inline void _read8(std::ifstream& fin, char& v){
    fin.read(reinterpret_cast<char*>(&v), sizeof(v));
}
inline void _readn(std::ifstream& fin, uint8_t* v, uint32_t n){
    fin.read(reinterpret_cast<char*>(v), n);
}
```

Then the program reads the binary data using `ifstream`, checks the correctness of the file and restores the information in `cv::Mat` objects.

(global definition)

```
void load_data(const char* imagefile, const char* labelfile,
               cv::Mat& images, cv::Mat& labels, const char* typ){
    printf("$ Loading %s data...\t", typ);

    std::ifstream fimg(imagefile, std::ios::binary);
    std::ifstream flab(labelfile, std::ios::binary);
    assertm(fimg.is_open(), "Cannot open image file.");
    assertm(flab.is_open(), "Cannot open label file.");

    uint32_t magic_img, magic_lab;
    uint32_t n, n_img, n_lab;
    uint32_t rows, cols;
    _read32(fimg, magic_img);
    _read32(fimg, n_img);
    _read32(fimg, rows);
    _read32(fimg, cols);
    _read32(flab, magic_lab);
    _read32(flab, n_lab);
    assertm(magic_img == 0x00000803, "Incorrect image file.");
    assertm(magic_lab == 0x00000801, "Incorrect label file.");
    assertm(n_img == n_lab, "The number of images and labels are not equal.");
    assertm(rows == 28 && cols == 28, "The size of images is not 28x28.");
    n = n_img;

    images = cv::Mat(n, rows*cols, CV_32FC1);
    labels = cv::Mat(n, 1, CV_32SC1);
    uint8_t *img = new uint8_t[rows*cols];
    uint8_t label;
    for(uint32_t i=0; i<n; i++){
        _readn(fimg, img, rows*cols);
        _read8(flab, label);
        for (uint32_t j = 0; j < rows * cols; j++)
            images.at<float>(i, j) = static_cast<float>(img[j]) / 255.0f;
        labels.at<int32_t>(i, 0) = static_cast<int32_t>(label);
    }
    delete[] img;
    fimg.close();
    flab.close();

    printf("Done.\n$ %d %s data have been loaded.\n", n, typ);
}
```

OpenCV provides handy ways to train and test a KNN model. In the program, the training part and the testing part are separated into two functions, and in each function it just calls OpenCV APIs to set up parameters and run the model. The accuracy is calculated by the correct predicts over the test samples.

(in main function)

```
train_knn(knn, train_images, train_labels);
test_knn(knn, test_images, test_labels);
```

(global definition)

```
void train_knn(cv::Ptr<cv::ml::KNearest>& knn,
               const cv::Mat& train_images, const cv::Mat& train_labels){
    printf("$ Training KNN model...\t");
    knn = cv::ml::KNearest::create();
    knn->setDefaultK(12);
    knn->setIsClassifier(true);
    knn->train(train_images, cv::ml::ROW_SAMPLE, train_labels);
    printf("Done.\n");
}
void test_knn(const cv::Ptr<cv::ml::KNearest>& knn,
              const cv::Mat& test_images, const cv::Mat& test_labels){
    printf("$ Testing KNN model...\t");

    cv::Mat pred;
    knn->predict(test_images, pred);
    pred.convertTo(pred, CV_32SC1);
    uint32_t n = test_images.rows, tot = 0;
    for(uint32_t i=0; i<n; i++)
        if(pred.at<int32_t>(i, 0) == test_labels.at<int32_t>(i, 0))
            tot++;
    printf("Done.\n$ Accuracy: %.2f%%\n", static_cast<float>(tot) / n * 100);
}
```

Finally the trained model is saved to mnist_knn.xml.

(in main function)

```
knn->save("mnist_knn.xml");
```

(3) applicable predictions

The trained model is first loaded.

(in main function)

```
knn = cv::ml::StatModel::load<cv::ml::KNearest>("mnist_knn.xml");
```

Then the program keeps requesting for image input. The input should be a valid path, corresponding

to a gray-scale picture (size is not required). The program will first normalize the picture then predicts the number. Users may input “sample1.png” (or so) to test using the sample images under the same folder.

Result presentations

Following are the screenshots of the program to help users check if they run the program correctly.

```
Welcome! This is the assignment 6 of CSC3002 by Chen Fu (122090020).
This program implements a KNN model to recognize handwritten digits.
The MNIST dataset is used to train and test the model.

A pre-trained model is provided. You can use it directly or train a new one.
Otherwise you may need about 4 mins to train a new model.
Use the pre-trained model? ([Y]/N)
```

Figure 1
Guideline Page

Figure 2
Sample Images
(with different sizes)

```
Input the path of the image to be recognized (or "exit" to exit): sample1.png
Predicted label: 2

Input the path of the image to be recognized (or "exit" to exit): sample2.png
Predicted label: 1

Input the path of the image to be recognized (or "exit" to exit): sample3.png
Predicted label: 5
```

Figure 3
Sample Outputs

The overall accuracy, as mention in last section, calculated by correct predicts over sample size, is 96.61% on the developer’s device. It almost approaches the best possible result without deep neural network on the MNIST data set, which is excellent.

The training time, less than five minutes on an Intel i5-8350U CPU, is faster than any other non-deep-network model with similar performance (e.g. SVM), which is excellent, too.

The image tested can be arbitrary size as long as it present a clear image of digit, since the program adapted resize function for any input. Other interactive sessions are dealt with carefully as well. Hence the program is robust enough to all primary cases of inputs.

Conclusion

In this project, a handwritten digit recognition algorithm is implemented using C++ language. By introducing OpenCV library, the program builds a KNN model with accuracy over 96.6% easily within a few hundred lines of code. The program also paid attention to robustness of the program.

Although there is still room for improvement, the result is enough satisfying.