C++ Project: Handwritten Digit Recognition with OpenCV

Xue Zhongkai 122090636

1. Introduction

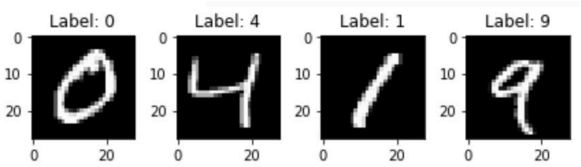
Handwritten digit recognition is a classic problem of computer vision (CV), dealing with the challenge of identifying and classifying handwritten digits from images.

It has a wide range of applications, such as reading checks for banks, sorting mail for postal services and processing hand-written forms for corporate sectors. With this technic, they could be extracted into the digital copy automatically, without relying on manual input.

At its core, handwritten digit recognition involves processing images of digits, extracting features, and then classifying them into one of the ten digit classes (0 to 9). The complexity arises from the vast variability in handwriting styles, sizes, and the quality.

This project aims to replicate the handwritten digit recognition, with:

- Algorithm: **Support Vector Machine (SVM)**
- Implementation Language: Mostly in **C++**
- General Package: **OpenCV**
- Traning/Test Dataset: Well-known **MNIST**



2. Project Structure

2.1 Source file organization:

```
/ (root)
|-- cmake-build-debug/ (directory)
|-- mnist_raw/ (directory)
|-- CMakeLists.txt
|-- output.txt
|-- predict.h
|-- read_mnist.h
|-- predict.cpp
|-- read_mnist.cpp
```

2.2 Replication of the project:

Here is the original project setup utilization:

- A **MacBook Pro (14-inch, 2021, M1-Pro chip)**, running the **Monterey 12.5** operating system.
- The IDE employed as **CLion (Version: 17.0.6+10-b829.9 x86_64)**.
- Locally pre-installed **OpenCV 4.8.1** via Homebrew.

The `CMakeLists.txt` file is included within the source files and can be compiled and run normally through CLion. The compilation folder `cmake-build-debug` is included within the source files.

Also you could execute bash commands as follows:

```
cmake .
make
./proj1
```

Important Note:

- This project is built on the **arm64 architecture** originally.
- If you fail to load the datasets, it might require changing the datasets' directories in `read_minst.cpp`.

3. Project Implementation

The entire project includes several parts, including **dataset reading**, **model training, prediction and evaluation**, and **saving the model files**. Each part has clear header and implementation files, with the headers providing concise comments for clients.

3.1 Dataset Reading

This component requires extensive knowledge of file handling, for which I consulted this source. The code is primarily responsible for reading and processing the MNIST dataset of handwritten digit images and labels, loading them from files, and converting them into a format suitable for machine learning operations. It includes functions for **byte order conversion**, **reading and transforming image data**, and **converting labels into one-hot encoding** (especially for `ANN` implementation).

Here I will put an emphasis on these two:

- `read_mnist_image`: This function reads the file containing handwritten digit images, parses the file header information, and then reads the image data pixel by pixel, converting it into **a single-row vector** `Mat` format.

  ```
  Mat read_mnist_image(const string& fileName);
  ```

- `read_mnist_label`: This function reads the file containing image labels, also starting by parsing the file header information, and then converts the label data into `Mat` format for subsequent processing.

  ```
  Mat read_mnist_label(const string& fileName);
  ```

3.2 Model Training and Evaluation

Following the official guidance from OpenCV, I randomly selected four machine learning models: **SVM**, **KNN**, **Decision Tree (DTree)**, and **Artificial Neural Network (ANN)**.

Before training, all the images are pre-processed for **normalisation**:

```
Mat images = read_mnist_image(images_path);
images = images / 255.0;
```

Each was **trained on the training set with a `<chrono>` timer**, and then used to compare labels on the test set to **calculate accuracy**.

Due to space constraints, I will only provide a detailed introduction to the parametres of SVM:

- The SVM type is set to `C_SVC`, a variant used for **classification tasks**.
- The kernel function is defined as `polynomial`, which is crucial for handling **non-linear datasets**.
- The `Gamma` and `Degree` parameters are set, vital for the **performance of the polynomial kernel**.
- The training stops when either the **maximum number of iterations** is reached or a preset **accuracy threshold** is met.

```
svm->setType(cv::ml::SVM::C_SVC);
svm->setKernel(cv::ml::SVM::POLY);
svm->setGamma(3.0);
svm->setDegree(3.0);
svm->setTermCriteria(cv::TermCriteria(cv::TermCriteria::MAX_ITER | cv::TermCriteria::EPS, 300,
0.0001));
```

After the model training is complete, predictions are made on the test set. Following format conversion, the predicted results are **compared one by one** with the labels, and finally, the prediction accuracy is outputted.

```
int equal_nums = 0;
for (int i = 0; i < pre_out.rows; i++) {
  if (pre_out.at<uchar>(i, 0) == test_labels_converted.at<uchar>(i, 0)) {
    equal_nums++;
  }
}
float acc = float(equal_nums) / float(pre_out.rows);
```

Ultimately, we save the model as an `XML` file (in `cmake-build-debug` directory), which is beneficial for **analyzing the model size**. This will determine its suitability for large-scale deployment.

```
svm->save("mnist_svm.xml");
```

Important Note:

- The full terminal output of the project is accessible as `output.txt`.
- Other three algorithms are implemented in the similar way.

4. Project Results

After several adjustments of the parametres, here are the output results:

	Training Time	Prediction Accuracy	Model Size
SVM	32.0881s	97.55%	30.3MB
KNN	0.0174539s	97.05%	268.8MB
DTree	18.5415s	54.6%	1.9MB
ANN	28.714s	96.26%	1.4MB

This summary could be analyzed as follows:

- SVM offers the **best prediction accuracy** with **a reasonable model size**.

  Its model size is moderate, demonstrating a balance between good performance and relatively low resource requirements.

- KNN offers the **fastest speed**, but has a large model size which may not be suitable for large-scale deployment.

  Actually KNN does not require an explicit training process; it classifies by examining the k-nearest neighbors during the testing phase. However, its model size is quite large because it needs to store the entire training dataset.

- DTree, with the lowest accuracy, is NOT suitable for this project.

  We could either try models like **Random Forest** or do more on **data pre-processing**.

- ANN, while slightly less accurate than SVM, has **the smallest model size**, which could be advantageous for deployment in resource-constrained environments.

  It adjusts a large number of weights and biases through the backpropagation algorithm, but indicates an application of a highly optimized and compressed network structure.

Important Note:

- The results are based on my hardware and software device, but offer a reference for comparison.
- The results could still be improved via adjusting parameters.
