香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

C/C++ Programming (CSC3002)

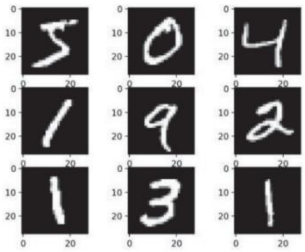
121040028 – Bryan Delton Tawarikh Sibarani –Assignment 6 Project

Introduction

Hand digit recognition has become increasingly crucial in the modern world, finding practical application in our daily lives. These systems enable us to tackle more intricate problems with greater ease. Automatic processing of bank checks and postal addresses exemplifies the widespread use of hand-written digit recognition. In this specific task, we trained Artificial Neural Network models to identify handwritten digits ranging from 0 to 9.

In a neural network, a node can be likened to a neuron in the brain, with each node interconnected to others through weights (essentially the edges between the nodes) that are adjusted within the algorithm. Every node's value is determined based on the features and methods of the preceding node; a process known as forward propagation. The final output of the system is then compared with the desired output, and weights are adjusted according to the loss function to indicate whether the system has effectively predicted the outcome. This process is referred to as backpropagation. Neural networks incorporate complexity and accuracy through various layers. In a fully-connected neural network, different layers, including input, output, and hidden layers, contribute to enhancing the system's capabilities.

Datasets and Tasks



Comprising 60,000 images distributed across 10 classes (0-9), MNIST dataset is notably extensive. Each image within the dataset measures 28 pixels in height and 28 pixels in width, resulting in 784-dimensional vectors. Accessible online, the MNIST dataset consists of grayscale images, with pixel values ranging from 0 to 255, representing the brightness and darkness of individual pixels.

Code Implementation

We know that from the official website http://yann.lecun.com/exdb/mnist/, both of the training set and testing set images and labels are stored in the IDX file format which need to be extracted first before data processing. After we extract the data, we can do the classification task using ML model. Therefore, the codes are divided based on two steps.

1. Data Extraction and Preprocessing (mnist_utils.cpp and mnist_utils.h)
   This code module has three different functions that works together to extract the image data.
   a. readIDX3UByteFile

This function is responsible for reading IDX3-UBYTE image files. The input of the function is “filename”, a string specifying the path to the IDX3-UBYTE file.

1) File Opening in Binary Mode
2) Header Reading: The function reads the header information of the IDX3-UBYTE file, including the magic number, the number of images, the number of rows, and the number of columns.
3) Image Reading: A vector is initialized to store the images. Then, for each image in the file, a vector of unsigned characters is created and filled by reading the corresponding number of bytes from the file. The image vector is added to the vector of images.

This function will output a vector of vectors, where each inner vector represents an image stored as a vector of unsigned characters.

b. readLabelFile

This function is designed to read IDX1-UBYTE label files associated with images. It follows a similar structure to the `readIDX3UByteFile` function but specifically reads label information. The input is also “filename”, which is a string specifying the path to the IDX1-UBYTE label file.

1) File Opening in Binary Mode
2) Header Reading: The function reads the header information of the IDX1-UBYTE label file, including the magic number and the number of labels.
3) Label Reading: A vector is initialized to store the labels. For each label in the file, a vector of a single unsigned character is created and filled by reading one byte from the file. The label vector is added to the vector of labels.

This function will output a vector of vectors, where each inner vector represents a label stored as a vector of unsigned characters.

c. readMNISTFiles

This function is designed to read MNIST dataset files, which typically consist of images and corresponding labels. The function utilizes two functions defined before, then extracting image and label information and organizing them into vectors of OpenCV matrices (cv::Mat) and integers, respectively. There are 4 input parameters, which are a string specifying the path to the MNIST image file, a string specifying the path to the MNIST label file, a vector of OpenCV matrices to store the image data, and a vector of integers to store the label data.

1) File Reading: The function calls the readIDX3UByteFile, obtaining a vector of vectors of unsigned characters (imagesFile), then calls the readLabelFile, obtaining another vector of vectors of unsigned characters (labelsFile).
2) Image Processing: For each image in the dataset, the function iterates over the corresponding vector of unsigned characters (imagesFile[imgCnt]). It constructs an OpenCV matrix named ‘tempImg’ with a size of 28x28 pixels and initializes it with zeros. The unsigned characters are mapped to pixel values in the matrix tempImg, row by row, column by column. The constructed image matrix is then added to the ‘imagesData’ vector.

3) Label Processing: For each image in the dataset, the function retrieves the label value from the corresponding vector of unsigned characters (labelsFile[imgCnt]) and converts it to an integer. The integer label is then added to the 'labelsData' vector.
4) Visualization (commented): The function contains commented code for visualizing images in testing. It uses OpenCV to display each image in dataset.

2. Deep Learning Algorithm (dnn_utils.cpp and dnn_utils.h)
   This module includes two functions: trainingModel and testModelAccuracy. These functions are designed to train a deep neural network (DNN) model using OpenCV's multi-layer perceptron module (cv::ml::ANN_MLP) and evaluate its accuracy.
   a. trainingModel
      This function is responsible for constructing, training, and returning a deep neural network (DNN) model using the provided training dataset. The input parameters are a vector of OpenCV matrices containing training image data and a vector of integers containing corresponding training labels.
      1) DNN Model Architecture: An instance of `cv::ml::ANN_MLP` is created to represent the DNN model. The architecture is defined with an input layer size (784 which is flattened of 28x28 image), a hidden layer size (in here a 100 hidden layers), and an output layer size (10 layers indicating probability for each digit). The activation function is set to be the sigmoid function.
      2) Training Data Preparation: Input trainingData is reshaped and converted to a CV_32F matrix. Labels (labelData) are one-hot encoded for classification.
      3) Model Training: DNN model is trained using the backpropagation algorithm with a specified termination criteria and learning rate. The cv::ml::TrainData class is used to create training data for the model. Then the model is trained for like 5-10 minutes for one hidden layer.

      At the end, this function will return the trained DNN model.

   b. testModelAccuracy
      This function evaluates the accuracy of a trained DNN model on a given dataset. The input parameters are a pointer to the trained DNN model, a vector of OpenCV matrices containing test image data, and a vector of integers containing corresponding test labels. For each test image, the model predicts the class, then the predicted class is compared to the actual label. If the prediction is correct, the true prediction count is incremented. The overall accuracy is calculated as the percentage of correct predictions over the total number of test samples.
      In the end, this function will return the calculated accuracy in floating number.
      The function contains commented code (`// To display test image and prediction...`) for visualizing predictions during testing. It uses OpenCV to display each test image and its predicted class

Reproduce Project

The project contains two header files (mnist_utils.h and dnn_utils.h), three cpp files (mnist_utils.cpp, dnn_utils.cpp, main.cpp), a folder mnist_data containing datasets, and three trained models in xml file. Mainly, to reuse the project, the users can do several things.

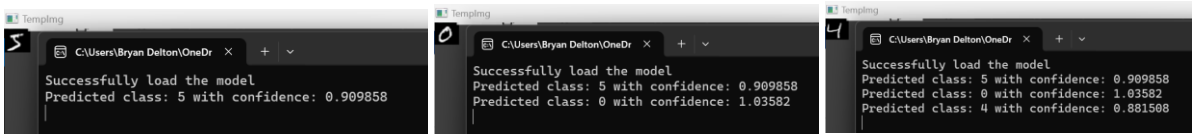
First, in the main.cpp, line 22-24 which is the model training is commented in default since the project already provide the load trained model in line 28. To build your own model, you can uncomment the line 22-24, name your own model, and comment the line 28. You can customize your own model (hyperparameter, number of layers, etc.) by modifying the code part at dnn_utils.cpp in the function trainingModel.

Second, you can also use different trained model by modifying the line 28. For example, you can load trained_mnist_model_2HU_100.xml where the architecture builds on 2 hidden layers 100x100 units or trained_mnist_model_800HU.xml where the architecture built on 1 hidden layer 800 units. You can also uncomment the visualization part in testModelAccuracy to see prediction of the model for each image test case.

After done with the modification, user can build the project and compile it to be main.exe. Make sure the compiler settings already include all OpenCV libraries and compile all header reference mnist_utils.h and dnn_utils.h including its own cpp code. At the end, when you run the main.exe file, the code will return the training accuracy and test accuracy of the model.

Results

Here is example result in testing the model by uncommenting the visualization part.



In this task, there are three developed model with different architectures that have the result as follows.

Model	Trained Data Accuracy	Test Data Accuracy
1 Hidden Layer 100 units	98.42%	96.78%
1 Hidden Layer 800 units	99.72%	97.08%
2 Hidden Layer 100+100 units	99.33%	97.25%

Conclusions

The experiments on various configurations of neural network models for the MNIST dataset reveal crucial insights.

1. Hidden Layer Size Impact: Increasing hidden layer size significantly boosts training accuracy but has a marginal effect on test accuracy, suggesting a risk of overfitting.
2. Multiple Hidden Layers Impact: Two hidden layers offer a modest improvement in test accuracy, indicating a potential for better generalization compared to a single larger hidden layer.
3. Trade-off Between Complexity and Generalization: Model complexity must be carefully considered to strike a balance between high training accuracy and effective generalization to new data.

In general, neural networks already work really good in this classification task of MNIST models. Even with the simplest model, we can achieve outstanding test accuracy.