```cpp
// The contents of this file are in the public domain. See LICENSE_FOR_EXAMPLE_PROGRAMS.txt
/*
    This is an example illustrating the use of the deep learning tools from the
    dlib C++ Library.  In it, we will train the venerable LeNet convolutional
    neural network to recognize hand written digits.  The network will take as
    input a small image and classify it as one of the 10 numeric digits between
    0 and 9.

    The specific network we will run is from the paper
        LeCun, Yann, et al. "Gradient-based learning applied to document recognition."
        Proceedings of the IEEE 86.11 (1998): 2278-2324.
    except that we replace the sigmoid non-linearities with rectified linear units.

    These tools will use CUDA and cuDNN to drastically accelerate network
    training and testing.  CMake should automatically find them if they are
    installed and configure things appropriately.  If not, the program will
    still run but will be much slower to execute.
*/


#include <dlib/dnn.h>
#include <iostream>
#include <dlib/data_io.h>

using namespace std;
using namespace dlib;

int main(int argc, char** argv) try
{
    // This example is going to run on the MNIST dataset.
    if (argc != 2)
    {
        cout << "This example needs the MNIST dataset to run!" << endl;
        cout << "You can get MNIST from http://yann.lecun.com/exdb/mnist/" << endl;
        cout << "Download the 4 files that comprise the dataset, decompress them, and" << endl;
        cout << "put them in a folder.  Then give that folder as input to this program." << endl;
        return 1;
    }


    // MNIST is broken into two parts, a training set of 60000 images and a test set of
    // 10000 images.  Each image is labeled so that we know what hand written digit is
    // depicted.  These next statements load the dataset into memory.
    std::vector<matrix<unsigned char>> training_images;
    std::vector<unsigned long>         training_labels;
    std::vector<matrix<unsigned char>> testing_images;
    std::vector<unsigned long>         testing_labels;
    load_mnist_dataset(argv[1], training_images, training_labels, testing_images, testing_labels);


    // Now let's define the LeNet.  Broadly speaking, there are 3 parts to a network
    // definition.  The loss layer, a bunch of computational layers, and then an input
    // layer.  You can see these components in the network definition below.
    //
    // The input layer here says the network expects to be given matrix<unsigned char>
    // objects as input.  In general, you can use any dlib image or matrix type here, or
    // even define your own types by creating custom input layers.
    //
    // Then the middle layers define the computation the network will do to transform the
    // input into whatever we want.  Here we run the image through multiple convolutions,
    // ReLU units, max pooling operations, and then finally a fully connected layer that
    // converts the whole thing into just 10 numbers.
    //
    // Finally, the loss layer defines the relationship between the network outputs, our 10
    // numbers, and the labels in our dataset.  Since we selected loss_multiclass_log it
    // means we want to do multiclass classification with our network.   Moreover, the
    // number of network outputs (i.e. 10) is the number of possible labels.  Whichever
    // network output is largest is the predicted label.  So for example, if the first
    // network output is largest then the predicted digit is 0, if the last network output
    // is largest then the predicted digit is 9.
    using net_type = loss_multiclass_log<
                                fc<10,
                                relu<fc<84,
                                relu<fc<120,
                                max_pool<2,2,2,2,relu<con<16,5,5,1,1,
                                max_pool<2,2,2,2,relu<con<6,5,5,1,1,
                                input<matrix<unsigned char>>
                                >>>>>>>>>>>>;
    // This net_type defines the entire network architecture.  For example, the block
    // relu<fc<84,SUBNET>> means we take the output from the subnetwork, pass it through a
    // fully connected layer with 84 outputs, then apply ReLU.  Similarly, a block of
    // max_pool<2,2,2,2,relu<con<16,5,5,1,1,SUBNET>>> means we apply 16 convolutions with a
    // 5x5 filter size and 1x1 stride to the output of a subnetwork, then apply ReLU, then
```

```
        // perform max pooling with a 2x2 window and 2x2 stride.



        // So with that out of the way, we can make a network instance.
        net_type net;
        // And then train it using the MNIST data.  The code below uses mini-batch stochastic
        // gradient descent with an initial learning rate of 0.01 to accomplish this.
        dnn_trainer<net_type> trainer(net);
        trainer.set_learning_rate(0.01);
        trainer.set_min_learning_rate(0.00001);
        trainer.set_mini_batch_size(128);
        trainer.be_verbose();
        // Since DNN training can take a long time, we can ask the trainer to save its state to
        // a file named "mnist_sync" every 20 seconds.  This way, if we kill this program and
        // start it again it will begin where it left off rather than restarting the training
        // from scratch.  This is because, when the program restarts, this call to
        // set_synchronization_file() will automatically reload the settings from mnist_sync if
        // the file exists.
        trainer.set_synchronization_file("mnist_sync", std::chrono::seconds(20));
        // Finally, this line begins training.  By default, it runs SGD with our specified
        // learning rate until the loss stops decreasing.  Then it reduces the learning rate by
        // a factor of 10 and continues running until the loss stops decreasing again.  It will
        // keep doing this until the learning rate has dropped below the min learning rate
        // defined above or the maximum number of epochs as been executed (defaulted to 10000).
        trainer.train(training_images, training_labels);

        // At this point our net object should have learned how to classify MNIST images.  But
        // before we try it out let's save it to disk.  Note that, since the trainer has been
        // running images through the network, net will have a bunch of state in it related to
        // the last batch of images it processed (e.g. outputs from each layer).  Since we
        // don't care about saving that kind of stuff to disk we can tell the network to forget
        // about that kind of transient data so that our file will be smaller.  We do this by
        // "cleaning" the network before saving it.
        net.clean();
        serialize("mnist_network.dat") << net;
        // Now if we later wanted to recall the network from disk we can simply say:
        // deserialize("mnist_network.dat") >> net;


        // Now let's run the training images through the network.  This statement runs all the
        // images through it and asks the loss layer to convert the network's raw output into
        // labels.  In our case, these labels are the numbers between 0 and 9.
        std::vector<unsigned long> predicted_labels = net(training_images);
        int num_right = 0;
        int num_wrong = 0;
        // And then let's see if it classified them correctly.
        for (size_t i = 0; i < training_images.size(); ++i)
        {
            if (predicted_labels[i] == training_labels[i])
                ++num_right;
            else
                ++num_wrong;

        }
        cout << "training num_right: " << num_right << endl;
        cout << "training num_wrong: " << num_wrong << endl;
        cout << "training accuracy:  " << num_right/(double)(num_right+num_wrong) << endl;

        // Let's also see if the network can correctly classify the testing images.  Since
        // MNIST is an easy dataset, we should see at least 99% accuracy.
        predicted_labels = net(testing_images);
        num_right = 0;
        num_wrong = 0;
        for (size_t i = 0; i < testing_images.size(); ++i)
        {
            if (predicted_labels[i] == testing_labels[i])
                ++num_right;
            else
                ++num_wrong;

        }
        cout << "testing num_right: " << num_right << endl;
        cout << "testing num_wrong: " << num_wrong << endl;
        cout << "testing accuracy:  " << num_right/(double)(num_right+num_wrong) << endl;

    }
    catch(std::exception& e)
    {
        cout << e.what() << endl;
    }
```