Laboratory Exercise 5

Classification of Handwritten Digits using Machine Learning: CNN-Based Classifier

In this exercise you will implement a convolutional neural network (CNN) based classifier of handwritten digits. You will create software and hardware implementations of a neural network classifier that accepts images of handwritten digits ranging 0 to 9 and determines which digit is present in each image.

The MNIST Handwritten Digits Database

In this exercise you will use the MNIST handwritten digits database, which is widely used by scientists around the world to train and test handwritten digit recognition machines. The database provides 28x28-pixel 8-bit grayscale images, each containing a single handwritten digit ranging 0 to 9. An example of an image from the MNIST database is shown in Figure 1.

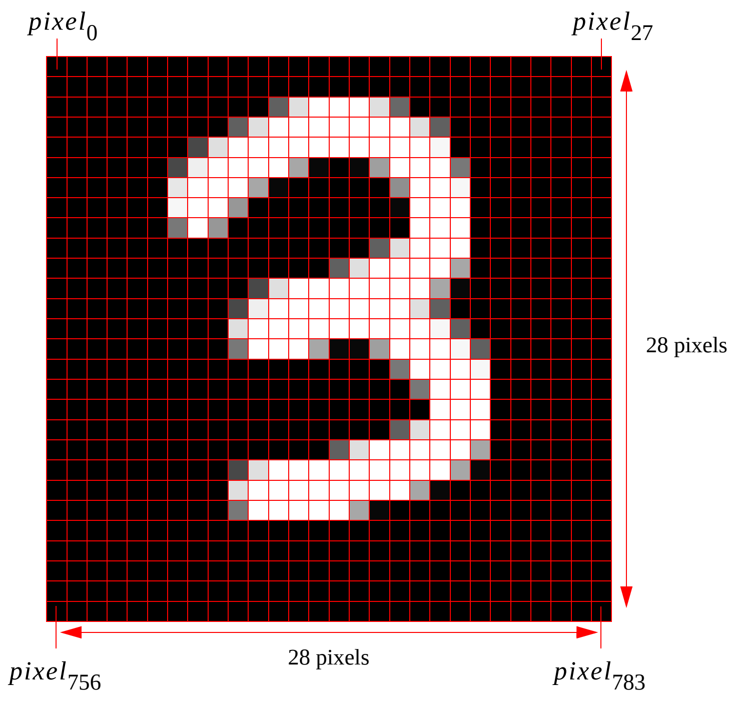


Figure 1: An example handwritten digit image from the MNIST database

The database contains a training set of 60000 images which are used to train the classifier, and a test set of 10000 images which are used to test the classifier's accuracy. The 60000 training images have already been used to train the weights for the classifier. Your task is to use the provided weights to implement the classifier and test its classification accuracy on the 10000 test images.

A Simple CNN-Based Classifier for MNIST Digits

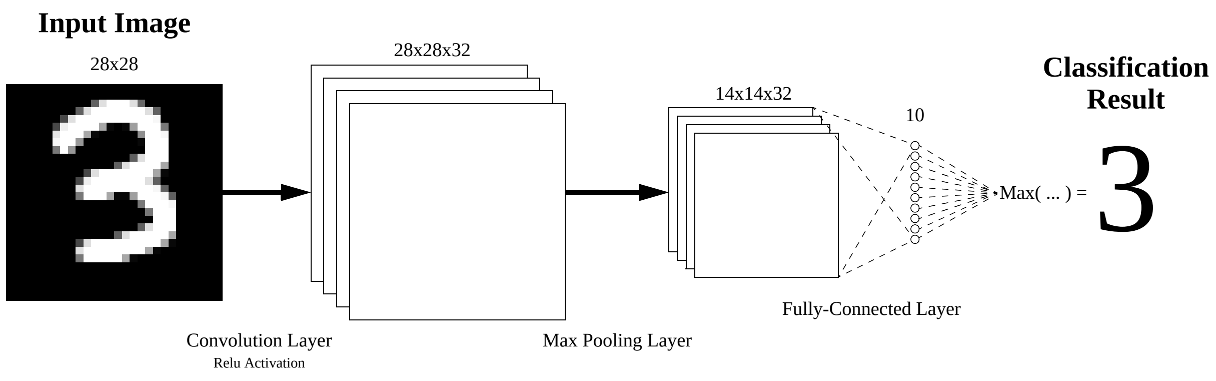
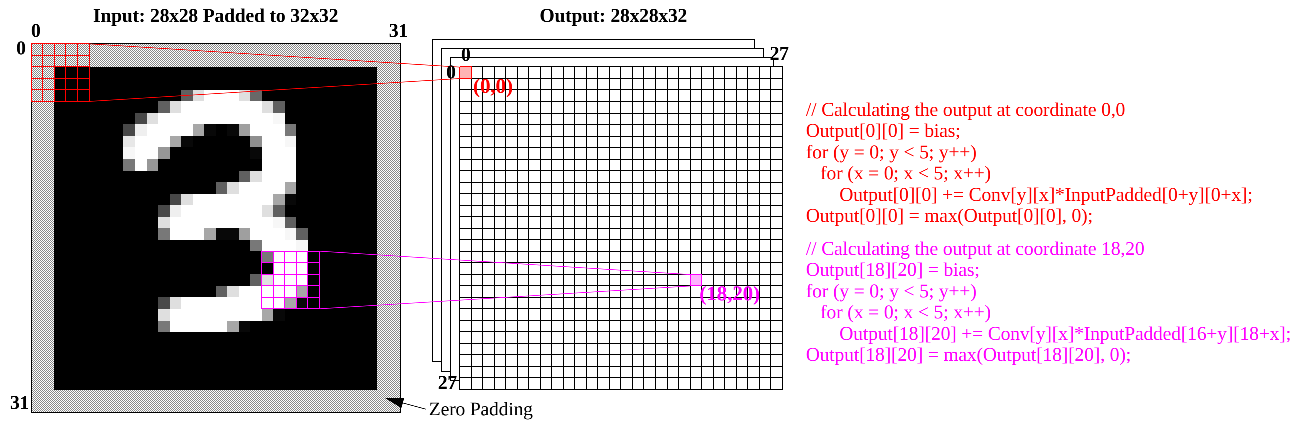


Figure 2: The layers of the CNN-based classifier

You will implement the CNN-based Classifier shown in Figure 2. The classifier accepts an MNIST digit image as input which has the dimensions 28x28. The classifier provides as output the classification result, ranging 0 to 9. Between the input and output layers, the classifier contains three hidden layers which are described in the following sections.

The Convolution Layer



```
// Calculating the output at coordinate 0,0
Output[0][0] = bias;
for (y = 0; y < 5; y++)
  for (x = 0; x < 5; x++)
    Output[0][0] += Conv[y][x]*InputPadded[0+y][0+x];
Output[0][0] = max(Output[0][0], 0);

// Calculating the output at coordinate 18,20
Output[18][20] = bias;
for (y = 0; y < 5; y++)
  for (x = 0; x < 5; x++)
    Output[18][20] += Conv[y][x]*InputPadded[16+y][18+x];
Output[18][20] = max(Output[18][20], 0);
```

Figure 3: Applying a convolutional matrix to the input image with calculations shown for two of the output values

The convolution layer applies 32 5x5 convolutional matrices on the input image to yield 32 feature maps. Before convolving, the layer adds 2-pixel zero padding to all edges of the input image so that the output map retains the same dimensions as the input image. Each convolutional matrix is applied to 5x5 patches of the padded image using a stride of 1 pixel. Each convolutional matrix has an associated trained bias value which is added to each convolution result. The ReLU activation function (defined as ReLU(x) = x > 0 ? x : 0) is then applied to the biased convolution result. The ReLU-activated value is then written to the output feature map. Figure 3 shows a convolutional matrix being applied to the input image to yield one feature map. The figure shows the calculations involved in producing two of the values in the output feature map. In the end, the 32 convolutional matrices yield 32 28x28 feature maps for a total output with dimensions 28x28x32.

The Max Pooling Layer

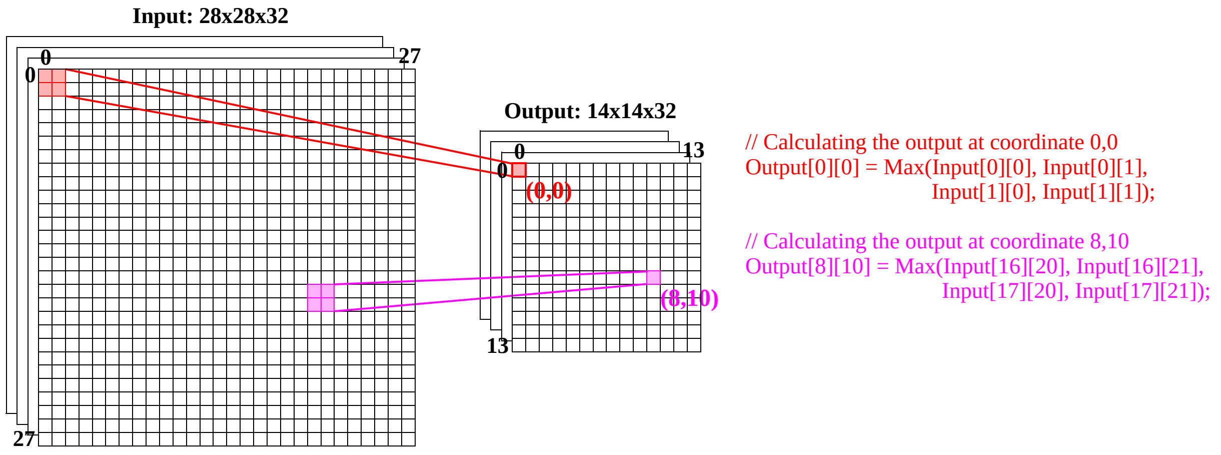

Input: 28x28x32
0
0
27
27
Output: 14x14x32
0
0
13
13
(0,0)
(8,10)
// Calculating the output at coordinate 0,0
Output[0][0] = Max(Input[0][0], Input[0][1], Input[1][0], Input[1][1]);
// Calculating the output at coordinate 8,10
Output[8][10] = Max(Input[16][20], Input[16][21], Input[17][20], Input[17][21]);

Figure 4: Applying the 2x2 max pooling on a 28x28 input with calculations shown for two of the output values

The max pooling layer downsamples the feature maps produced by the CNN layer by retaining only the maximum value in each 2x2 block of values. Downsampling in this layer reduces the number of features in flight and thus the number of calculations required in the subsequent layers. It also reduces the amount of memory required to store subsequent features. By applying 2x2 max pooling with a stride of 2 pixels on a 28x28x32 input, the layer produces an output with dimensions 14x14x32.

The Fully Connected Layer

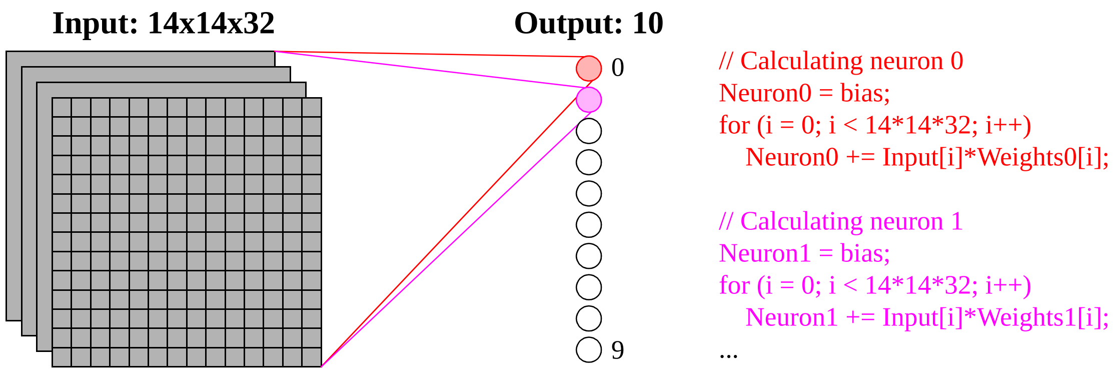

Input: 14x14x32
Output: 10
0
9
// Calculating neuron 0
Neuron0 = bias;
for (i = 0; i < 14*14*32; i++)
Neuron0 += Input[i]*Weights0[i];
// Calculating neuron 1
Neuron1 = bias;
for (i = 0; i < 14*14*32; i++)
Neuron1 += Input[i]*Weights1[i];
...

Figure 5: The fully connected layer with calculations shown for two of the output neurons

The fully connected layer yields 10 output neurons, each of which is calculated as the weighted sum of the 14x14x32 features output from the previous layer. This means that in total there are 14x14x32x10 = 62720 weights. In essence, this fully-connected layer is a linear classifier whose input are the features outputted by the max pooling layer, and output are the likelihoods of the input image belonging to each of the 10 digit classes. The digit with the highest output value is chosen as the classification result.

Part I

Implement a program in the C++ language that performs CNN-based classification of MNIST handwritten digits as described in the previous section. Use the trained weights provided in the */design_files/weights_fxp/* directory. Test your program by classifying the 10000 MNIST test images stored in the file */design_files/t10k-images-idx3-ubyte*. Compare the results of your classifier to the correct labels stored in the file */design_files/t10k-labels-idx1-ubyte* to calculate your classifier's accuracy as:

$$Accuracy = (\#\ of\ correct\ results)/10000 * 100\%$$

Compile and execute your program on a DE-Series Linux platform. Upon completion, your program should output the classification accuracy and the runtime in milliseconds.

Parsing the MNIST Database Files

The MNIST database is provided in .idx files. The file formats of the MNIST images and labels files are shown in Figure 6. Your program must open and parse these files to extract the image pixels and labels. Each pixel is 8 bits and represents a grayscale value. Each label is also 8 bits, and its value can range from 0 to 9 to indicate the correct classification of the corresponding image. The 32-bit integers stored in these files are stored in most-significant byte (MSB)-first format, which must first be converted to least significant byte (LSB)-first format before being used on the ARM processors of the DE-series boards.

```
TEST SET IMAGE FILE (t10k-images-idx3-ubyte):
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000803(2051) magic number
0004     32 bit integer  10000            number of images
0008     32 bit integer  28               number of rows
0012     32 bit integer  28               number of columns
0016     unsigned byte   ??               pixel
0017     unsigned byte   ??               pixel
........
xxxx     unsigned byte   ??               pixel

TEST SET LABEL FILE (t10k-labels-idx1-ubyte):
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000801(2049) magic number (MSB first)
0004     32 bit integer  10000            number of items
0008     unsigned byte   ??               label
0009     unsigned byte   ??               label
........
xxxx     unsigned byte   ??               label
The labels values are 0 to 9.
```

Figure 6: File formats of the MNIST files. Source: http://yann.lecun.com/exdb/mnist/

Parsing the Weights Files

Trained neural network weights are provided in the directory */design_files/weights_fxp/*. The weights are in 16-bit (8.8) fixed-point format. In the directory are 10 files named *fc_weights_0*, *fc_weights_1*, ... , *fc_weights_9* that provide the weights for the 10 neurons of the fully-connected layer. The digit for which the file provides the weights is suffixed to the file name. The weights in these files are arranged row-first, starting with weight corresponding to coordinate 0,0 (top-left) of feature map 0 (the output map of convolutional matrix 0). Finally there is one weights file named *cnn_weights* which provide the weights for the 32 5x5 convolutional matrices of the CNN layer. The *cnn_weights* file contains 32 sets of 26 weights, where the first 25 of each set of 26 correspond to the 5x5 matrix weights, and the 26th is the bias. The file stores the sets starting from convolutional matrix 0. The 5x5 matrix weights are arranged row-first starting from coordinate 0,0 (top-left) of the 5x5 matrix. The files can be parsed using the code shown in Figure 7.

```
// weights must be an array of sufficient size (>= num_weights)
bool read_weights_file(char *filename, short *weights, int num_weights) {
    FILE *f = fopen(filename, "rb");
    if (f == NULL){
        printf("ERROR: could not open %s\n",filename);
        return false;
    }
    int read_elements = fread(weights, sizeof(short), num_weights, f);
    fclose(f);

    if (read_elements != num_weights){
        printf("ERROR: read incorrect number of weights from %s\n", filename);
        return false;
    }
    return true;
}
```

Figure 7: Function that parses a weights file

Part II

Create an OpenCL implementation of the CNN-based classifier you created in Part I. You may wish to reuse code from your linear classifier in Exercise 4 to implement the fully-connected layer. Consider whether you will implement a single monolithic OpenCL kernel or divide the work into multiple smaller kernels. If implementing multiple kernels, decide whether to use global memory or pipes to share the data. Carefully consider how you will utilize the limited local memory resources to maximize performance. Finally, output the accuracy and runtime of your implementation and compare them to your results in Part I.

Part III (Bonus)

Devise a neural network topology for a classifier that improves on the accuracy you achieved in Part II. Create an OpenCL implementation of your classifier. As you will be limited by the amount of resources available to you, carefully consider the tradeoffs between accuracy, runtime, and FPGA resource usage. The parameters that you can adjust include:

- Number of CNN layers, convolutional matrix size(s), number of matrices per layer, stride size
- Number of pooling layers, size, and stride
- Number of FC layers, and number of intermediate neurons
- Precision (8, 16, 32 bit fixed-point, decimal position, mixed precision)

To train weights you can use a machine learning framework of your choice. Some common frameworks include Tensorflow and Caffe.