University of Hong Kong

Department of Computer Science

COMP8502 - Advanced Topics in Pattern Recognition

Final Project Report

TITLE:

Handwritten Digits Recognition using Convolutional Neural Network

GROUP MEMBER:

NAME: Ao Eerdemotai HKUID: 2012951823

NAME: He Rui HKUID: 2012951976

1. Introduction

In our final project, we implement the convolutional neural network for handwritten digits recognition. The MNIST database is used both in training and testing, and in order to accelerate the process, we apply GPU programming skills into the previous CPU algorithm with the help of C++ AMP. Experiment with a subset of the training dataset has been made, which works out an advisable recognition ratio 73% in 200 training epochs. Further improvement is needed in variable learning rate, pooling technique and more efficient GPU utilization, etc.

2. Terminology

2.1 Convolutional Neural Network

Convolutional Neural Networks (CNNs) are variants of Neural Networks, which exploit spatially local correlation by enforcing a local connectivity pattern between neurons of adjacent layers. In CNNs, each sparse filter is additionally replicated across the entire visual field. These “replicated” units form a feature map, which share the same weight vector and the same bias. Conceptually, a feature map is obtained by convolving the input image with a linear filter, adding a bias term and then applying a non-linear function.

2.2 C++ AMP

C++ Accelerated Massive Parallelism (C++ AMP) accelerates execution of C++ code by taking advantage of data-parallel hardware such as a graphics processing unit (GPU) on a discrete graphics card. The C++ AMP programming model includes multidimensional arrays, indexing, memory transfer, tiling, and a mathematical function library.

2.3 MNIST

The MNIST database of handwritten digits, established by Yann LeCun and Corinna Cortes, has a training set of 60,000 examples, and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed-size 28×28 image of gray values.

3. Methodology

3.1 Structure of the Convolutional Neural Network

Input Data:

The input is the grayscale image of the handwritten character from the MNIST database. The original 28×28 image is padded to 29×29 pixels due to the consideration of the structure of Layer #0.

Layer #0:

Layer #0 is a convolutional layer with 6 feature maps. Each feature map is sized to 13×13 pixels/neurons. Each neuron in each feature map is a 5×5 convolutional kernel of the input layer, but every other pixel of the input image is skipped. As a consequence, there are 13 positions where the 5×5 kernel will fit in each row of the input image, and 13 positions where the 5x5 kernel will fit in each column of the input image. There are therefore $13 \times 13 \times 6 = 1014$ neurons, and $(5 \times 5 + 1) \times 6 = 156$ weights ("+1" is for the bias) in Layer #0.

Layer #1:

Layer #1 is also a convolutional layer, but with 50 feature maps. Each feature map is 5×5, and each unit in the feature maps is a 5×5 convolutional kernel of corresponding areas of all of the 6 feature maps of the previous layer, each of which is a 13×13 feature map. There are therefore $5 \times 5 \times 50 = 1250$ neurons, and $(5 \times 5 \times 6 + 1) \times 50 = 7550$ weights in Layer #1.

Layer #2:

Layer #2 is a fully-connected layer with 100 units. Since it is fully connected, each of the 100 neurons in the layer is connected to all 1250 neurons in the previous layer. There are therefore 100 neurons, $100 \times (1250 + 1) = 125100$ weights, and $100 \times (1250 + 1) = 125100$ connections in Layer #2.

Layer #3:

Layer #3 is the final, output layer. This layer is a fully-connected layer with 10 units. Since it is fully-connected, each of the 10 neurons in the layer is connected to all 100 neurons in the previous layer. There are therefore 10 neurons, $10 \times (100 + 1) = 1010$ weights, and $10 \times (100 + 1) = 1010$ connections in Layer #3.

3.2 Forward Propagation

Forward propagation is the process whereby each neuron calculates its output value, based on inputs provided by the output values of the neurons that feed it, according to the feed-forward formula, namely:

$$x_n^i = F(y_n^i) = F(\sum_{k=0}^{C_{n-1}} \omega_n^{ik} \cdot x_{n-1}^k)$$

In the above formula, $F(.)$ refers to the Activation Function, which squash the input value into range of [-1, 1]. $F(y_n^i) = \tanh(y_n^i)$ is chosen due to the convenience of obtaining its derivative, $G(y_n^i) = 1 - x^2$.

This formula is applied by iterating through all connections for the neuron, and for each connection, obtaining the corresponding weight and the corresponding output value from a neuron in the previous layer.

After 4 forward iterations, reaching the output Layer #3, exactly one neuron has a value of +1 corresponding to the answer (hopefully), while all other nine neurons have an output of -1.

3.3 Back Propagation

Back propagation is an iterative process that starts with the last layer and moves backwards through the layers until the first layer is reached. Start the process off by computing the partial derivative of the error due to a single input image pattern with respect to the outputs of the neurons on the last layer.

The error due to a single pattern is calculated as follows:

$$E_n^P = \frac{1}{2} \cdot \sum (x_n^i - T_n^i)^2 \qquad \text{(Equation 1)}$$

Given the above error equation, we can derive that:

$$\frac{\partial E_n^P}{\partial x_n^i} = x_n^i - T_n^i \qquad \text{(Equation 2)}$$

$$\Rightarrow \frac{\partial E_n^P}{\partial y_n^i} = G(x_n^i) \cdot \frac{\partial E_n^P}{\partial x_n^i} \qquad \text{(Equation 3)}$$

$$\Rightarrow \frac{\partial E_n^P}{\partial \omega_n^{ij}} = x_{n-1}^j \cdot \frac{\partial E_n^P}{\partial y_n^i} \qquad \text{(Equation 4)}$$

$$\Rightarrow \frac{\partial E_n^P}{\partial x_{n-1}^k} = \sum_{k=0}^{C_{n-1}} \omega_n^{ik} \cdot \frac{\partial E_n^P}{\partial y_n^i} \qquad \text{(Equation 5)}$$

The numeric values obtained from Equation 5 are used as starting values for the calculations on the immediately preceding layer, and we compute

Equation 3, 4 and 5 in a repetition for the back propagation.

Meanwhile, the values from Equation 4 tell us how much to change the weights in the current layer n. In particular, we update the value of each weight according to the formula:

$$(\omega_n^{ij})_{new} = (\omega_n^{ij})_{old} - eta \cdot (\frac{\partial E_n^P}{\partial \omega_n^{ij}})$$

Eta is the learning rate, which is set as a constant 0.001 in our program.

3.4 C++ AMP acceleration

Use the **accelerator** and **accelerator_view** classes to specify the device or emulator to run C++ AMP code on.

```
accelerator defaultDevice(accelerator::default_accelerator);
accelerator_view defaultView = defaultDevice.default_view;
```

Use **array_view** class to create C++ AMP objects. Data passed to the array_view constructor is copied to the accelerator when the kernel function is executed.

```
array_view<float,2>   inputlayer_output_GPU(...);
array_view<float,1>   layer0_weights_GPU(...);
array_view<float,3>   layer0_output_GPU(...);
array_view<float,1>   dE_dx_layer0_GPU(...);
array_view<int,1>     layer0_kernel_GPU(...);
```

The C++ AMP code that we want to run on the accelerator is specified as an argument in a call to the **parallel_for_each** method. Either a lambda expression or a function object could be that argument. Additionally, the

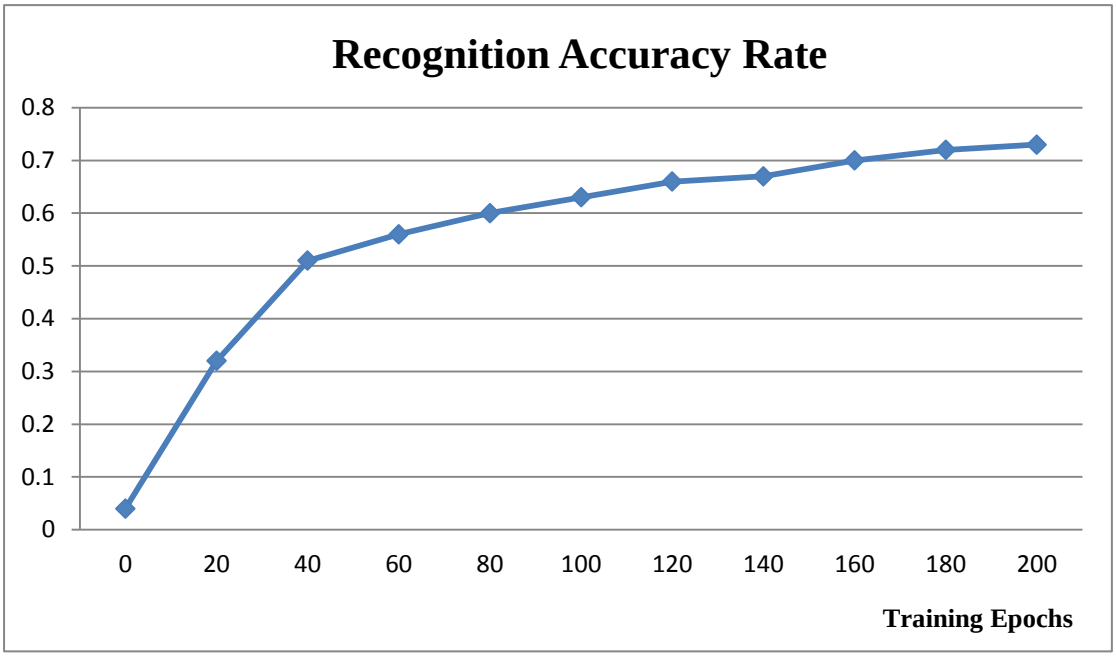
lambda expression or function object can call a C++ AMP-restricted function.

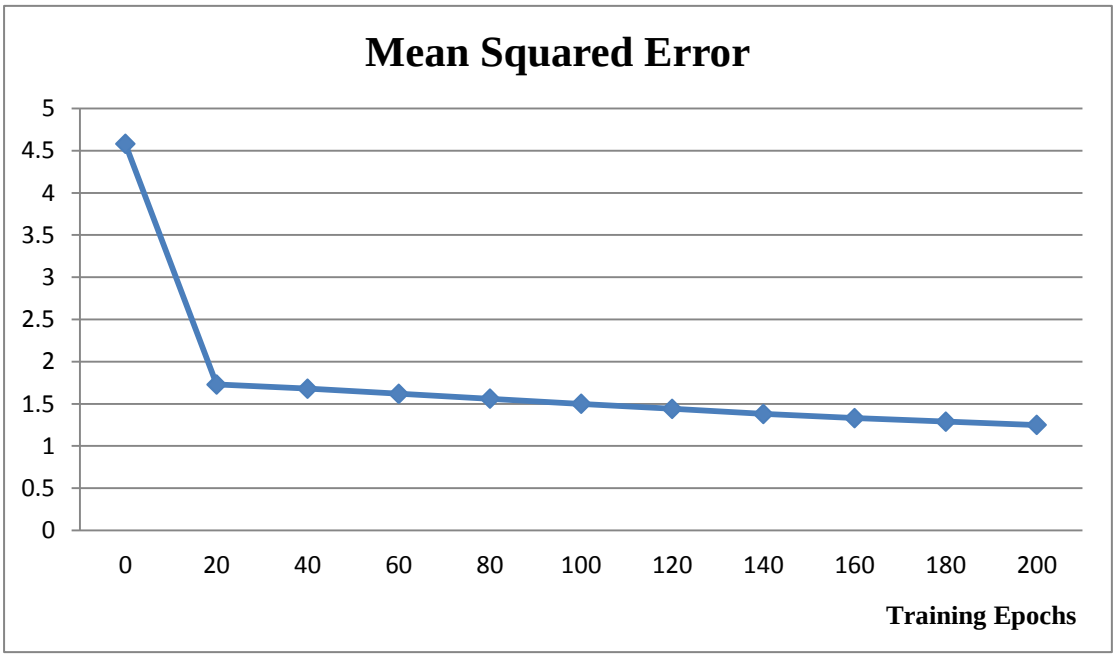
```
parallel_for_each(
        layer0_output_GPU.extent,
        [=](index<3> idx) restrict(amp)
{
   //...
   //... complete task

    layer0_output_GPU(map_index,row,column)=current_output;
}
layer0_output_GPU.synchronize();
```

4. Experimental Results

From 60000 training samples of the MNIST database, we choose 100 samples to complete the experiment, observing the change of the recognition ratio and mean squared error with the increasing of the training epochs. The result is as follows:





In 200 training epochs, we achieve the advisable recognition accuracy rate 73.00% and the mean squared error drops down to 1.25.

5. Discussion

In our program, the learning rate remains constantly as 0.001, which should be gradually decreased during the training to achieve a better performance. For convenient implementation, we connect adjacent layers directly by keeping a constant convolutional window index matrix. To attain more accurate results, we need use max-pooling technique, a form of non-linear down-sampling, to reduce the computational complexity for upper layers and provide a form of translation invariance. Though we use GPU to accelerate the training process, the running speed does not increase significantly. There should be large space to improve the C++ AMP code and second order algorithm need to be considered as well.