HandWritten Digits Recognition Implementing Backpropagation Neural Networks on CUDA

Xiaoyang Wang, Xueying Liao

1 Introduction

Backpropagation algorithm is a mathematical model which works far faster than earlier approaches to learning, making it possible to use neural networks to solve the problems which was unsolvable in the previous years. The process of backpropagation is iterative, that it starts from the last layer and moves backwards to the first layer. It calculates the change in the weights of each layer, which involves a large number of vector and matrix operations.

In this project, we use CUDA parallel programming platform and run on Graphics Processing Units(GPUs). There are some libraries for neural networks on CUDA like cudaNN, but the thing we are interested in is how to build a library rather than implement an algorithm with library. So instead of treat the neural network as a black box, we do some exploration on how to invent our own wheels.

2 Background

Backpropagation Neural networks mainly has two independent steps, which is predicting and training. Forward propagation is a process where neural nodes calculate the output value based on inputs. Backpropagation examine the error in the output of the previous layer given the output of current layer.

2.1 Forward propagation

Basically a neural network is a widely used prediction model with hypothesis function h(x) with parameter theta. The data is loaded to the input layer and we use sigmoid function- g(x) as hypothesis function and parameter matrix theta to calculate the 'activation' for the next layer. After propagating from input layer, through all the hidden layer, the data finally hit the output layer, where we get the prediction result. This process is also called forward propagation.

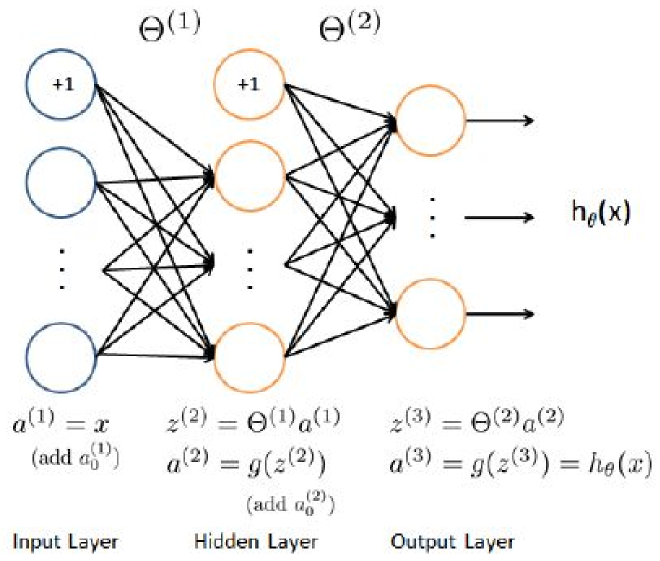


Fig. 1. Neural Network Model

The most important step in forward propagation is create a function to measure the accuracy. The possible value for each element in the output matrix are 0 and 1. For different reference(use y as reference) result, the error measurement is different, log(h(x)) for y = 1 and log(1 - h(x)) for y = 0. Shown below is the cost function we use to measure the error. The goal for back propagation is to minimize this cost function and make the most accurate prediction. And in the cost function, we implement regularization which is the sum of theta. This guarantees the value of theta is small and avoid overfitting problem.

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right] +$$
$$\frac{\lambda}{2m}\left[\sum_{j=1}^{25}\sum_{k=1}^{400}(\Theta_{j,k}^{(1)})^2 + \sum_{j=1}^{10}\sum_{k=1}^{25}(\Theta_{j,k}^{(2)})^2\right].$$

2.2 Backpropagation

After finishing forward propagation, for each node in each layer, we compute the error term "delta" that measures how much nodes were "responsible" for any error in our output. For the output layer, we have:

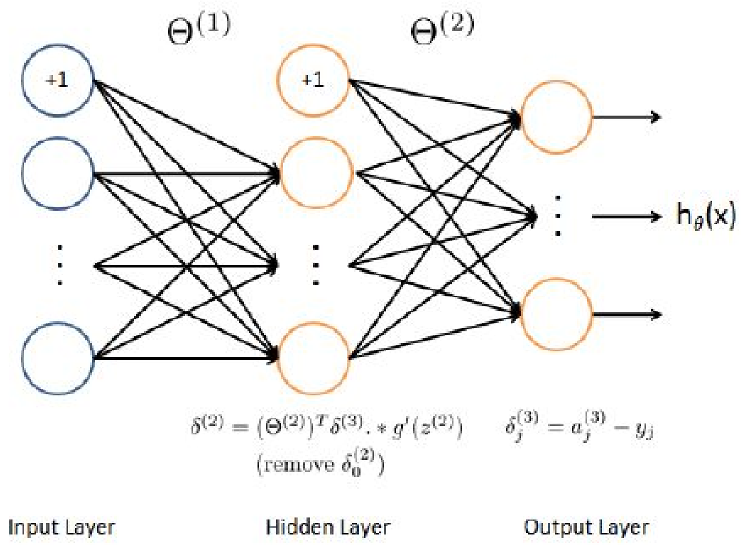
$$\delta_k^{(3)} = (a_k^{(3)} - y_k)$$

Where a_k is the output and y is the reference.

And for the other layer, we have:

$$\delta^{(2)} = \left(\Theta^{(2)}\right)^T \delta^{(3)} .* g'(z^{(2)})$$

Where g'(z) is the sigmoid gradient.



$\Theta^{(1)}$
$\Theta^{(2)}$
+1
+1
$h_\theta(x)$
$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$
(remove $\delta_0^{(2)}$)
$\delta_j^{(3)} = a_j^{(3)} - y_j$
Input Layer
Hidden Layer
Output Layer

Fig. 2. Backpropagate Updates

After obtain all error terms, we calculate the gradient based on these two functions:

$$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)}$$

Then after we get the partial derivative of each theta, we perform a gradient descent based on the derivatives. The Fig.3 below shows a three-dimensional diagram of gradient descent.

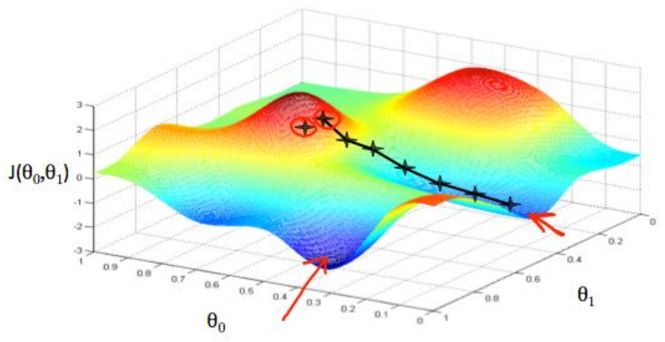


Fig. 3. Gradient Descent

If enough iterations is taken, the cost function will finally reach a local minimum, which is usually not worse than the global minimum.

3 Technique Description

3.1 The Structure of Backpropagation Used in This Paper

In this project, for each input sample, we make a one dimensional vector with 400 elements represent a 20x20 pixel picture of handwritten numbers. So we have 401 nodes in input layer(include one bias unit). And we have 26 nodes in the hidden layer(include one bias unit), as well as 10 nodes in the output layer, and each node stands for a number between 0 and 9. Thus, the size of parameter matrix theta1 is 401x25 and the size of parameter matrix theta2 is 26x10.

For output format, we need to reshape our reference vector. That is if we have an reference element equals to 8, which means the correct output should be 8, then we build a vector [0, 0, 0, 0, 0, 0, 0, 0, 1, 0] for this reference.

3.2 General Description of Backpropagation Algorithm

Backpropagation algorithm can be divided into two phases, one is a feed forward propagation to the output, and the other one is backpropagation phase that adjust the network weights. The fig.4 shown below is the framework of our project. We train the neural network in Training() function that handles the parallelism computation of the bias term, then pass them to the Predict() function that work on forward propagation again to calculate outputs which is much more accurate.

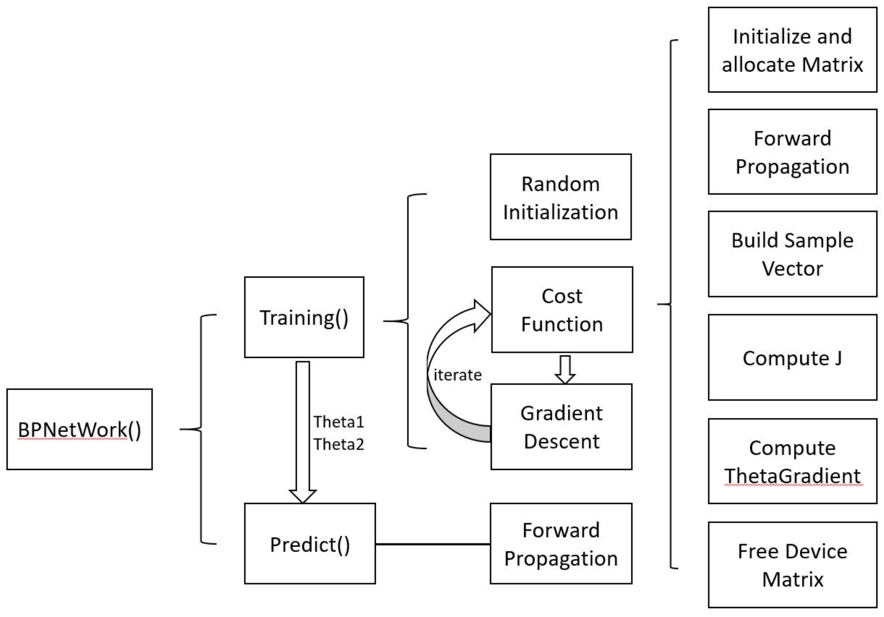


Fig. 4. Framework

4. Optimization

4.1 Build a tiny library using shared memory with tile

We build a tiny library that basically implements tile for all matrix operations if applicable. That is each block(tile, 32x32) fetch a 32x32 matrix from global memory, put it to the shared memory and pad one column to eliminate bank conflict.

4.2 Simplify calculation procedures

We write a function for forward propagation. In the original approach, this step is finished in five instructions as shown below.

```
a1 = X
a1 = pad one column for a1
Theta1 = Theta1'
z2 = a1*Theta1
a2 = sigmoid(z2)
```

But in this approach, we will waste a lot time and resources doing memory copy for padding column and matrix transpose. So in our implementation, we developed a faster approach, which treats the padding column as an independent vector. Figure First get a submatrix SubTheta1 from the original matrix theta1(in order to separate the first row) and do normal matrix multiplication for X and SubTheta1. Then do another matrix multiplication for padding column and the first row in Theta1. Also, we merge the matrix transpose to the matrix multiplication. The process of the optimization is described in Fig.5 and Fig.6. By doing this, four lines of code in sequential program are translated to one line in parallel program using CUDA.

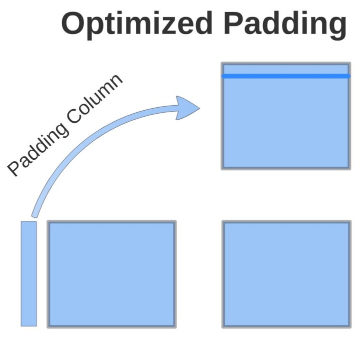


Fig. 5. Optimized Padding

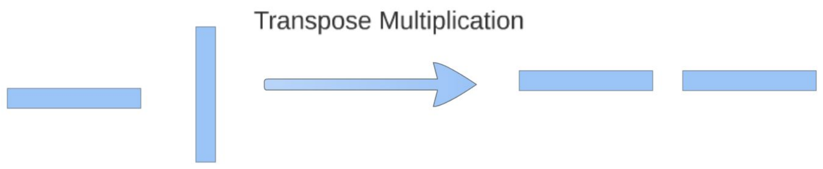


Fig. 6. Transpose Multiplication

4.3 Kill a for-loop

In the cost function, a for-loop is used to calculate Delta1 and Delta2. The Matlab version code is written as below.

```
for i = 1:m
    delta3 = a3(i, :) - y(i, :);
    delta2 = (delta3*Theta2)(:, 2:end) .* sigmoidGradient(z2(i, :));
    Delta2 += delta3'*a2(i, :);
    Delta1 += delta2'*a1(i, :);
endfor
```

After analyzing the code above, we find that if we unroll for-loop, the complex calculation on Delta1 and Delta2 can be viewed as matrix multiplication with transpose matrix. Thus, we implement a matrix multiplication on the CUDA platform without for-loop to obtain the value of Delta1 and Delta2.

4.4 Two dimensional sum

Sum reduction, especially two dimensional addition, is an important issue in CUDA that deserves discussion, since there are multiple skillful methods to save time. For maximum efficiency and easy implementation, we do sum reduction in shared memory. We calculate the sum of every row, which stores the value in the first column of each line. Then add the them together to get the final result. The kernel code of two dimensional sum is shown below. We intercept some of the code for the space limitation.

```
if(threadIdx.y < 2) {
      MTile[threadIdx.y][threadIdx.x] += MTile[threadIdx.y +
2][threadIdx.x];
      __syncthreads();
}

if(threadIdx.y < 1) {
      MTile[threadIdx.y][threadIdx.x] += MTile[threadIdx.y +
1][threadIdx.x];
      __syncthreads();
}

if(threadIdx.x < 2) {
      MTile[0][threadIdx.x] += MTile[0][threadIdx.x +  2];
```

```
        __syncthreads();
    }

    if(threadIdx.x < 1) {
        MTile[0][threadIdx.x] += MTile[0][threadIdx.x +  1];
        __syncthreads();
    }

    atomicAdd(result, MTile[0][0]);
```

5 Experimental results

For some reason, the overall performance cannot be estimated in this project. We tested a simple tiled matrix multiplication function on GPU with same function on CPU and get approximately 1.6 times speedup. But this is only a very small unit. If taking other finished and unfinished optimization into consideration, we assume we can get more speedup.

6 Conclusions

It is time consuming to calculate the weight of Neural Network using backpropagation. When applied on the CPU, it takes several minutes or hours. In this paper, we propose an idea to perform the complex calculation on CPU using CUDA. The calculation is proved to be able to run in parallel GPU, which is much more faster than run in sequential on CPU and several possible optimization can be implemented.

7 References

[1] NVIDIA Corporation(2012). CUDA C Programming Guide(Version 5.0)
[2] NVIDIA Corporation(2012). CUDA C Best Practices Guidd(Version 5.0)
[3] Kirk, B. D., & Hwu W. (2010). Programming Massively Parallel Processors. Burlington, MA: Elsevier Inc.