Deep Learning in openCL

NTU B99202064

June 27, 2015

1 Introduction

First of all, I would admit that the opencl in DNN involve too many of matrix operations that the is complex to deal with it and out of my expect that I plan to finish it in 1 days (and I spend three days I think). Hence, the result of opencl haven't been strictly checked that is correct and havent do the functionanlity of prediction (but in serial version it can predict) . The first half will introduce DNN and the second half will show some time comparison result between serial version and openCL version. Furthermore, we can combine openMp with openCL do fasten the process and the most important, is check the operation in kernel and the outcome is correct.

2 Objectives

Nowadays, deep learning arises quickly to become state of art in various AI areas such as computer vision[1], natural language processing[2], and speech recognition[3]. The main problems influence deep learning for a long time is the computation complexity and also how its algorithms quickly converge to a global optimum and avoid the local optimum. Recently, the GPU techniques well popular in accelerate deep learning training and finally to solve the DNN massive computation problem[4]. In this term project, I would like to approach the deep learning by solving the computation complexity using openMP/openCL to accelerate the computation speed. There are 2 ways to reach: 1. Using libgpuarray [5] of Theano to implement openCL version. Theano is a python library that using an abstract graph node to optimize its computation either in cpu or gpu. It is a powerful tool that when we using to implement deep learning. Again, it support CUDA but a little bit openCL. 2. Simply using openCL implement a type of deep neural network, just like CUDA do for deep neural network called cuDNN [6]. Furthermore, if we have a lot of CPU/GPU clusters, then we may have a parallel or distributed gradient descent training and even do the asynchronies gradient descent in DNN, there will be more fun to do this kind of experiments!

3 Methods and Algorithms

3.1 Supervised learning review

As figure 1 show that, a standard supervised learning framework will use a known data points vector that x is the data points and y is the labels to train a model, in the training procedure, we will define a error function that measure the difference between the model predict and the actual values. After that, we will miniminize the error function and let the model learn well in the training data.

3.2 Basics of Neural network

As figure 2 state that, the basic neural net structure is form by input layer, hidden layer with a lot of neuron, and a output layer.
Every neuron in a hidden layer has a transformation function θ

$$\theta(x) = tanh(x)$$

The neural network model is determined by architecture of the neural network, that the numer of nodes in each layer $d = [d^{(0)}, d^{(1)}, ..., d^{(L)}]$ where the L is the number of layers.
Let's zoom into a neuron in hidden layer l, as figure 3 state that, a node has an incoming signal s and an output x. The weights on links into the node from the previous layer are $w^{(l)}$,so the weights are indexed by the layer into which they go. Then, the output of the node is multiplied by weights w^{l+1}. The nodes in the input layer $l = 0$ are for the input values. So, the input to node j in layer l is $s_j^{(l)}$, and $w_{ij}^{(l)}$ is the weight into layer l from node i in the previous layer to node j in layer l. For convenience, we use matrix notation for the following equation.

layer l parameters

signals in	$s^{(l)}$	$d^{(l)}$ dimensional input vector
outputs	$x^{(l)}$	$(d^{(l)}+1)$ dimensional output vector
weights in	$W^{(l)}$	$(d^{(l-1)}+1)\times \mathrm{d}^{(l)}$ dimensional matrix
weights out	$W^{(l+1)}$	$(d^{(l)}+1)\times \mathrm{d}^{(l+1)}$ dimensional matrix

3.3 Forward Propagation

The final output is computer by the forward propagation algorithm. We can write down the relation between the inputs and outputs of a layer,

$$x^{(l)} = \begin{bmatrix} 1 \\ \theta(s^{(l)}) \end{bmatrix}$$

and we can specified weights in $W^{(l)}$: $s_j^{(l)} = \sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}$. In matrix form:

$$\mathbf{s}^{(l)} = (\mathbf{W}^{(l)})\mathbf{X}^{(l-1)}$$

Thus, we can write a forward propagation pseudo code that compute final output:
Forward Propagation:

```
1: $x^{(0)} \leftarrow x$                                  [Initialization]
2: for $l = 1$ to L do                                     [Forward Propagation]
3: ....$s^{(l)} \leftarrow (W^{(l)})^T X^{(l-1)}$
4: ....$x^{(l)} = \begin{bmatrix} 1 \\ \theta(s^{(l)}) \end{bmatrix}$
5: output $h(x) = X^{(L)}$                                 [Output]
```

3.4 Error function

Once we use forward propagation algorithm obtain the hypothesis $y_{predict} = h(x)$, then we can compute in sample error by using sum of squares error, Our goal is to learning all $w_{ij}^{(l)}$ to minimize $E_{in}(w_{ij}^{(l)})$

$$E_{in}(W) = \frac{1}{N}\sum_{n=1}^{N}(h(x_n) - y_n)^2$$

where $h(x_n)$ is predict value and y_n is actual value. Since E_{in} is a complex function and hard to find a closed-form solution, we will using gradient descent method to approach the solution.

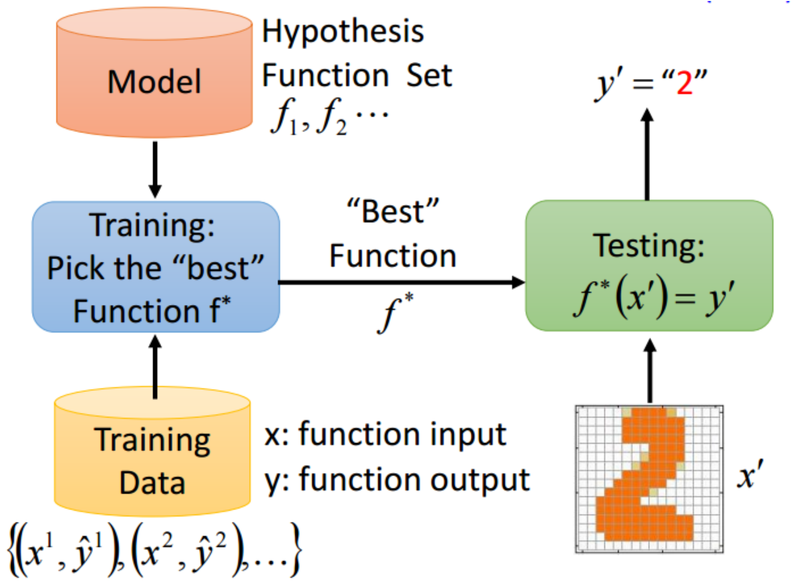


Figure 1: supervised learning

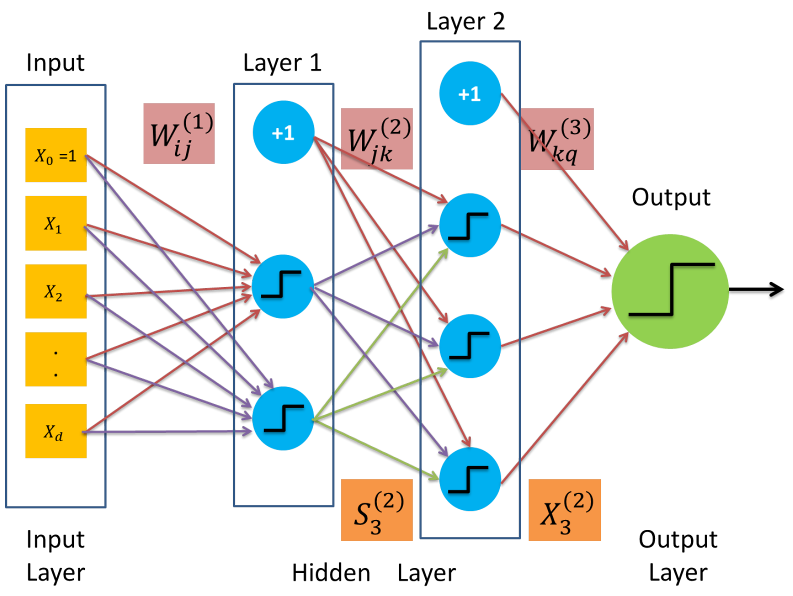


Figure 2: Feed Forward Neural Network

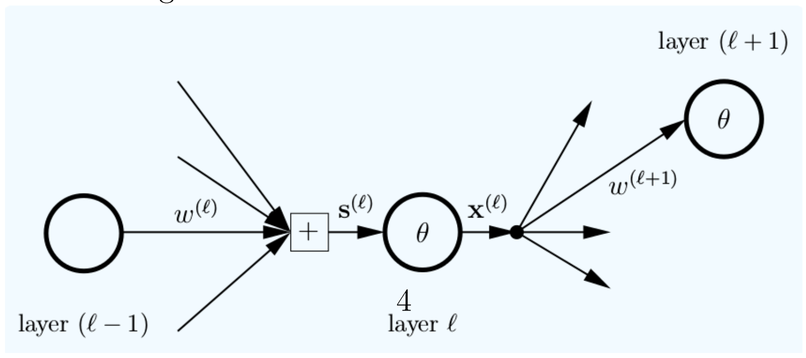


Figure 3: neuron structure

3.5 Backpropagation

The basics gradient descent method is to calculate the gradient of the function and approach to the minimum by every iteration step.
For simplicity, we first declare error function to be :

$$e_n = (y_n - s_1^{(L)})^2 = (y_n - \sum_{i=0}^{d^{(L-1)}} w_{i1}^{(L)} x_i^{(L-1)})^2)$$

where the error function is defined by the difference between output layer with actual y. By the chain rule, we can get:

$$\frac{\partial e_n}{\partial w_{ij}^{(l)}} = \frac{\partial e_n}{\partial s_j^{(l)}} \cdot \frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}}$$

and we define :

$$\delta_j^{(l)} = \frac{\partial e_n}{\partial s_j^{(l)}}$$

Then the equation become:

$$\frac{\partial e_n}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} \cdot (x_i^{(l-1)})$$

And for the output layer L:

$$\delta_1^{(L)} = -2(y_n - s_1^{(L)})$$

and error function for output layer:

$$\frac{\partial e_n}{\partial w_{ij}^{(l)}} = -2(y_n - s_1^{(L)}) \cdot (x_i^{(L-1)})$$

and the relation for $\delta^{(l)}$:

$$\delta^{(1)} \leftarrow \delta^{(2)} \leftarrow ... \leftarrow \delta^{(L-1)} \leftarrow \delta^{(L)}$$

and to calculate the gradient $\delta^{(l)}$:

$$\delta_j^{(l)} = \frac{\partial e_n}{\partial s_j^{(l)}} = \sum_{k=1}^{d^{(l+1)}} \frac{\partial e_n}{\partial s_k^{(l+1)}} \frac{\partial s_k^{(l+1)}}{\partial x_j^{(l)}} \frac{\partial x_j^{(l)}}{\partial s_j^{(l)}}$$

$$\delta_j^{(l)} = \sum_k (\delta_k^{(l+1)})(w_{jk}^{l+1})(tanh'(s_j^{(l)}))$$

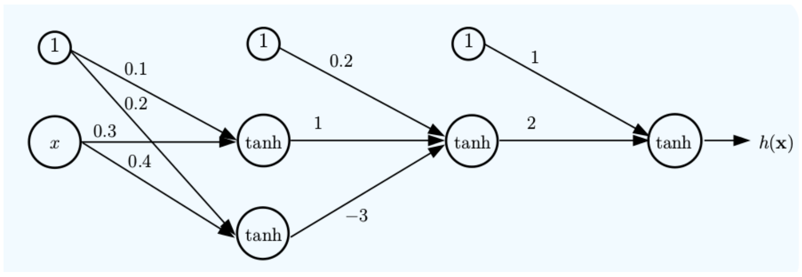
So that we can derive $\delta_j^{(l)}$ backwards from $\delta_k^{(l+1)}$
Backpropagation to compute $\delta_j^{(l)}$ pseudo code:
Assume $s^{(l)}$ and $x^{(l)}$ have been computed for all l

```
1:  $\delta^{(L)} \leftarrow \theta'(s^{(l)}) = -2(y_n - s_1^{(L)})$        [Initialization]
2:  for $l = L-1$ to 1 do                                  [Backward Propagation]
3:  ....$\delta_j^{(l)} \leftarrow (\delta^{(l+1)})(W^{l+1})(tanh'(s^{(l)}))$
```

3.6 Neural Net Algorithm

```
initialize all weights $w_{ij}^{(l)}$
for $t = 0, 1, ..., T$ [Iteration]
....1: stochastic: randomly pick $n \in \{1, 2, ..., N\}$
....2: forward: compute all $x_i^{(l)}$ with $x^{(0)} = x_n$ where $x_n$ is input data
....3: backward: compute all $\delta_j^{(l)}$ subject to $x^{(0)} = x_n$
....4: gradient descent: $w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta x_i^{(l-1)} \delta_j^{(l)}$
return $h(x) = (...tanh(\sum_j w_{jk}^{(2)} \cdot tanh(\sum_i w_{ij}^{(1)} x_i)))$
```

3.7 Toy Example



Consider above example,

$$W^{(1)} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix} ; W^{(2)} = \begin{bmatrix} 0.2 \\ 1 \\ -3 \end{bmatrix} ; W^{(3)} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Let $x = 2$, forward propagation to compute $h(x) = x^{(3)}$ gives:

$x^{(0)}$	$s^{(1)}$	$x^{(1)}$	$s^{(2)}$	$x^{(2)}$	$s^{(3)}$	$x^{(3)}$
$\begin{bmatrix}1\\2\end{bmatrix}$	$\begin{bmatrix}0.1 & 0.3\\0.2 & 0.4\end{bmatrix}\begin{bmatrix}1\\2\end{bmatrix}=\begin{bmatrix}0.7\\1\end{bmatrix}$	$\begin{bmatrix}1\\0.60\\0.76\end{bmatrix}$	$\begin{bmatrix}-1.48\end{bmatrix}$	$\begin{bmatrix}1\\-0.9\end{bmatrix}$	$\begin{bmatrix}-0.8\end{bmatrix}$	-0.66

We show above how $s^{(1)} = (W^{(1)})^T x^{(0)}$ is computed. Backpropagation gives

$\delta^{(3)}$	$\delta^{(2)}$	$\delta^{(1)}$
$\begin{bmatrix}0.56\end{bmatrix}$	$\begin{bmatrix}(1-0.9^2)\cdot 2\cdot 0.56\end{bmatrix} = \begin{bmatrix}0.21\end{bmatrix}$	$\begin{bmatrix}0.13\\-0.27\end{bmatrix}$

We show above how $\delta^{(2)}$ is computed from $\delta^{(3)}$. Then the gradient for update is easily to compute:

$$\frac{\partial h}{\partial W^{(1)}} = x^{(0)}(\delta^{(1)})^T = \begin{bmatrix}0.13 & -0.27\\0.26 & -0.54\end{bmatrix};\ \frac{\partial h}{\partial W^{(2)}} = \begin{bmatrix}0.21\\0.126\\0.157\end{bmatrix};\ \frac{\partial h}{\partial W^{(3)}} = \begin{bmatrix}0.56\\-0.504\end{bmatrix};$$

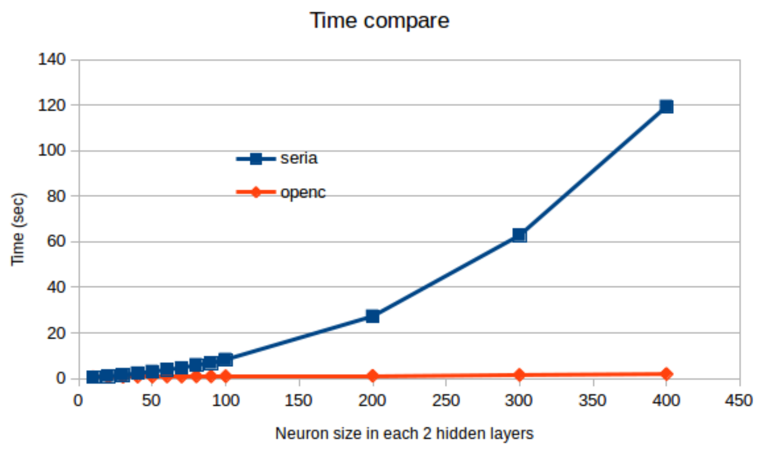
4 Application

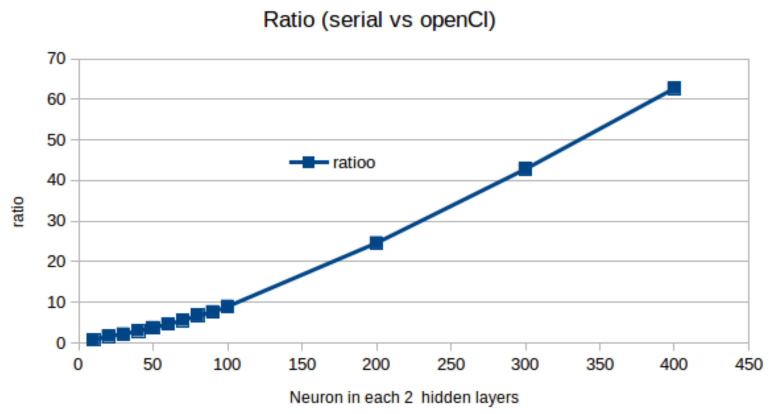
Since most of the research of deep learning in academy using CUDA to accelerate the multilayer neural network, but the problem of CUDA is just suitable for Nvidia only. By using openCL, we can make it more general to other like AMD, Intel even Nvidia. After that, that's a huge potential to build heterogeneous system [7] that combine several company GPUs by using general openCL, then will not limited by Nvidia but all vendors.

5 Expected outcome

In this term project, I expect that I can implement a simple version of DNN in openCL and can make it as an open source project to complete the openCL usage in DNN. After that, I can using this DNN version of openCL to do characters recognition or image classification to accelerate the computation speed and reach state of art result. Eventually, if time permit, then the asynchronies training via several clusters will be done.

6 Experiment and Result





In my DNN architecture, I implement 2 hidden layers DNN structure, with fully connected network, and in my experiment, I vary the neuron size in each of hidden layers, to increase the dimensionality of the matrix multiplication. The time is calculated by an forward propagation and backward propagation training epoch.

Hidden layer 1	Hidden layer 2	Serial Time	OpenCL Time
10	10	0.45525	0.629
20	20	0.984729	0.600191
30	30	1.41957	0.670978
40	40	2.28871	0.773928
50	50	2.96351	0.800416
60	60	3.83834	0.832198
70	70	4.56927	0.82611
80	80	5.91824	0.877797
90	90	6.75659	0.888779
100	100	8.26876	0.924332
200	200	27.279	1.10873
300	300	62.8155	1.4666
400	400	119.427	1.90453

7 Conclusion

The openCL result is better than serial version when matrix size increase by increase neuron size. The only doubt of the result maybe come from the bug of the kernel, since I don't have time to double check is the result is correct.

8 References

1. Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton (2012), Imagenet classification with deep convolutional neural networks, NIPS.
2. Thang Luong, Richard Socher, Christopher D. Manning. 2013. Better Word Representations with Recursive Neural Networks for Morphology. Conference on Computational Natural Language Learning (CoNLL 2013).
3. Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, Andrew Y. Ng (2014), Deep Speech: Scaling up end-to-end speech recognition. Techinical report, arXiv:1412.5567
4. Videos lecture, Large-scale Deep Unsupervised Learning using Graphics

Processors, Rajat Raina, Anand Madhavan, Andrew Y. Ng
videolectures.net/site/normal_dl/tag = 48368/*icml*09_*raina_lsd*_01.*pdf*
5. Theano, “Using the GPU”
http : //*deeplearning.net/software/theano/tutorial/using_gpu.html*
6. CUDA, cuDNN
https://developer.nvidia.com/cuDNN
7. Deep learning in openCL, BDTC 2014 China Big Data Conference
http : //*thumedia.org/assets/BDTC*/%*E*8%*B*0%*B*7%*E*4%*BF*%8*A*%*E*4%*B*8%*BD*%20−
%20%*E*5%9*F*%*BA*%*E*4%*BA*%8*E*%*E*5%*BC*%80%*E*6%94%*BE*%*E*6%*A*0%87%*E*5%87%86*Open*
8. Abu-Mostafa, Magdon-Ismail, Lin: Oct-2014, Learning from data, e-chap7, most of the calculation and example come from this book. 9. Tutorial: OpenCL SGEMM tuning for Kepler, http://www.cedricnugteren.nl/tutorial.php?page=7
10. A Hands-on Introduction, Simon McIntosh-Smith, University of Bristol
, http://ircc.fiu.edu/sc13/OpenCL01$_{S}$*lides.pdf*