Parallel Backpropagation for Multilayer Neural Networks

Nitin Kamra NKAMRA@USC.EDU
Department of Computer Science, University of Southern California

Palash Goyal PALASHGO@USC.EDU
Department of Computer Science, University of Southern California

Sungyong Seo SUNGYONS@USC.EDU
Department of Computer Science, University of Southern California

Vasileios Zois VZOIS@USC.EDU
Department of Computer Science, University of Southern California

Abstract

We present a parallel approach to classification using neural networks as the hypothesis class. Neural networks can have millions of parameters and learning the optimum value of all parameters from huge datasets can be very time consuming, if implemented on a single processor. In this work, we have implemented parallel backpropagation to train Multi Layer Perceptrons (MLPs) for classification tasks. Specifically, we implement a serial backpropagation algorithm and its parallel counterpart with Pthread library in C++. We also compare the two implementations with CUDA implementation of backpropagation on a GPU and study the parallelization speedup obtained for various network architectures and increasing problem sizes. We perform our tests on two benchmark datasets: MNIST and KDD Cup 1999, and finally compare all our implementations with the corresponding implementations in the state-of-the-art deep learning library: Theano.

1. Introduction

Artificial neural networks are powerful machine learning tools used in many applications including but not limited to search engines, spam and fraud detection, image classification, diagnostic medicine applications and stock market prediction.

Prior to application, neural networks undergo a training phase which is known to be very computationally intensive. This is primarily because the prevailing neural network architectures are implemented using several hidden layers, with each one consisting of thousands to millions of neurons in order to generalize well on diverse inputs. In this case, the resulting number of parameters that need to be trained are in the order of millions.

Furthermore, achieving high accuracy requires considering a large number of training examples (usually in the order of millions). For this reason training a neural network is both a data and resource intensive operation. This calls for efforts to parallelize the training process on multi-core and while emphasizing at utilizing the maximum available bandwidth.

Currently Minibatch Gradient Descent (henceforth called MGD) is the most commonly used optimization algorithm used to train neural networks in supervised settings. It is implemented in a layerwise-recursive fashion which is termed *backpropagation* in the context of neural networks.

In this paper, we have implemented parallel backpropagation to train Multi Layer Perceptrons (MLPs) for classification tasks. The rest of the paper is organized as follows: section 2 presents a description of supervised learning tasks and neural networks as classifiers. Section 3 describes the conventional serial backpropagation algorithm, and an approach to parallelization using multiple threads. Section 4 describes how to implement backpropagation on a GPU with cuda. Section 5 describes our datasets and the experiments we performed on them. Section 6 presents our results obtained for these implementations and analyzes the speedup obtained for various network architectures and increasing problem sizes. It also presents a comparison with the same algorithms implemented using a state-of-the-art neural network library Theano. Finally, we conclude in section 7 with a discussion of potential applications and future work for this project.

CSCI-503: Parallel Programming, Fall 2016, Final Project.

2. Problem Description

We first formally describe a classification task in the supervised learning setting. Then we describe a feedforward neural network with fully connected layers. Feedforward neural networks act as function approximators in such tasks.

2.1. Classification Problem

More formally, given a dataset $\mathcal{D} = \{x^{(i)}, y^{(i)}\}_{i=1:N}$ with data points $x^{(i)} \in \mathbb{R}^D$ and labels $y^{(i)} \in \mathbb{R}^P$, the classification task involves making an accurate label prediction $\hat{y}$ on a previously unseen data point x. We approximate the label as a function of the datapoint using a classifier (a feedforward neural network here) with parameters $\theta = \{\theta_k\}_{k=1:K}$ as follows: $y \approx \hat{y} = f(x; \theta)$. The classifier (neural network) learns the function f from the training data $\mathcal{D}$ by tuning its parameters (θ) to minimize a pre-specified loss function, for instance, the Mean-Squared Error loss:

$$\mathcal{L}_{MSE}(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - f(x^{(i)}; \theta))^2 \tag{1}$$

This minimization can be carried out using optimization algorithms like Gradient Descent, Newton's method, Levenberg-Marquardt algorithm etc. Though Newton's method is a second-order optimization technique, it requires the computation of the hessian of the objective function which is very prohibitive for a large number of parameters like in a neural network. Levenberg-Marquardt algorithm also requires computing matrices of the size of hessian and can be very slow for classifiers with a large number of parameters. Currently Gradient Descent is the most successful technique to train huge neural networks with millions of parameters, since it provides a decent tradeoff between convergence speed and memory requirements. We will describe gradient descent in section 3.

2.2. Feedforward Neural Networks

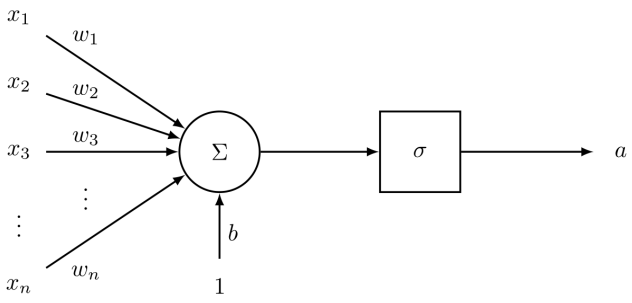

Figure 1. A neuron

The basic unit of feedforward neural networks is a neuron (figure 1). A single neuron generally takes a vector of inputs $x \in \mathbb{R}^n$ and outputs a single scalar $a \in \mathbb{R}$ (called its activation). Generally the function computed by a neuron comprises of linear transformation on the input vector followed by a pointwise non-linear activation function:

$$a = f(w^T x + b) \tag{2}$$

where $w \in \mathbb{R}^n$ is called the weight vector and $b \in \mathbb{R}$ is the scalar bias of the neuron. Many different kinds of activation functions have been studied and used according to the task at hand e.g. linear, sigmoid, tanh, ReLU etc. We will use the sigmoid and linear($f(z) = z$) activation functions in our implementation.

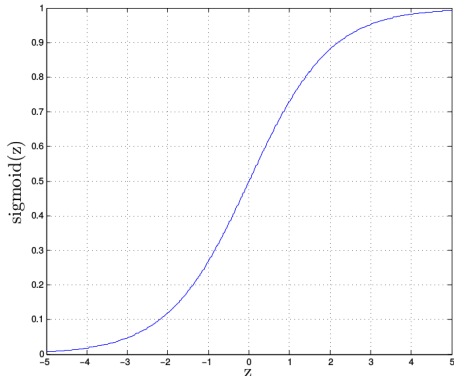

Figure 2. The sigmoid function

The sigmoid function is shown in figure 2 and is defined as follows:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3}$$

A feedforward neural network is a directed acyclic graph $G = (V, E)$ each of whose vertices $v \in V$ is a neuron and every edge $(u, v) \in E$ represents the output of neuron u going as an input to neuron v. In general to have a more concrete structure the graph G is organized into a layered structure and we will only use layered feedforward neural networks in our implementations.

A feedforward network with L layers comprises of a single input layer, $L-2$ hidden layers (their outputs are not directly observed) and a final output layer. Let the input of the neural network be $x \in \mathbb{R}^{n_{in}}$ and the output be $y \in \mathbb{R}^{n_{out}}$. The l^{th} layer has n_l neurons and the full network has $n = \sum_{l=1}^{L} n_l$ neurons. An example network is shown in figure 3.

The input to the first layer (denoted x_1) is x and the output (activation) of the first layer is denoted $a_1 \in \mathbb{R}^{n_{in}}$. The input layer contains $n_1 = n_{in}$ dummy neurons each of which takes a single scalar input n_{in} and passes it as it is i.e. $a_1 = x_1 = x$.

All subsequent layers $l \in \{2, 3, ..., L\}$ give an output $a_l \in \mathbb{R}^{n_l}$ and take an input $x_l = a_{l-1} \in \mathbb{R}^{n_{l-1}}$. The final network output y is given by $a_L \in \mathbb{R}^{n_L} (n_L = n_{out})$. Each of these

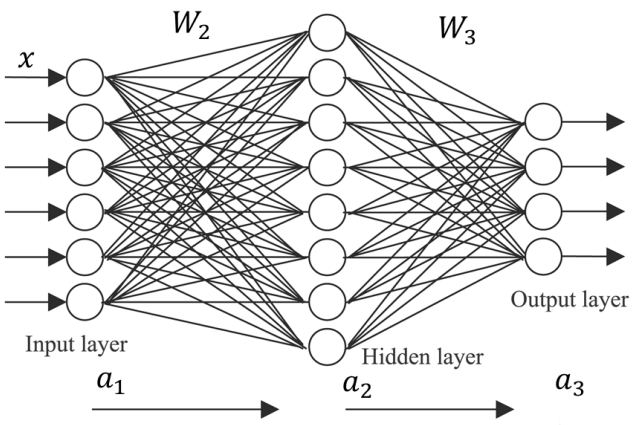

Figure 3. A neural network with one hidden layer

layers applies a linear transformation to its input followed by a pointwise non-linear activation function i.e. for each layer:

$$z_l = (W_l)^T x_l + b_l \tag{4}$$

$$a_l = f(z_l) \tag{5}$$

where $W^l \in \mathbb{R}^{n_{l-1} \times n_l}$ is the weight matrix of the layer and $b_l \in \mathbb{R}^{n_l}$ is the bias vector. The function $f(\cdot)$ is the pointwise activation function which applies to each component of the z_l vector and we will use sigmoid as our activation function for the classification task.

3. Gradient Descent and Backpropagation

As explained in section 2.1, classifiers try to choose their parameters (θ) in order to minimize some pre-specified loss function. In this work, we will work with the Mean-squared Error loss function as defined in equation 1.

3.1. Batch Gradient Descent

To optimize the loss function, we will use the Gradient Descent algorithm which in its most basic form takes the derivative of the cost function w.r.t. all the parameter values and updates the parameters by a value proportional to this gradient and opposite in sign. In its most basic form the Gradient Descent algorithm is implemented as shown in algorithm 1.

3.2. Minibatch Gradient Descent

Note that the derivative of $\mathcal{L}_{MSE}$ in algorithm 1 requires us to sum over all the training examples. This makes gradient computation, the biggest bottleneck for many supervised learning tasks on huge data sets. One solution to this problem is to approximate the gradient with a smaller batch of training examples selected from the full dataset, so that more updates can be made in a smaller amount of time. The resulting algorithm is called Minibatch Gradient Descent (abbreviated as MGD) and is shown in algorithm 2.

Algorithm 1 Batch Gradient Descent

Input: Dataset $\mathcal{D} = \{x^{(i)}, y^{(i)}\}_{i=1:N}$, Step Size α, Max Epochs N_{epoch}
Output: Parameters $\{\theta_k\}_{k=1:K}$

Initialize θ randomly with small real numbers
$ep := 0$
repeat
  $ep := ep + 1$
  for $k \in \{1 : K\}$ **do**
    $\frac{\partial \mathcal{L}_{MSE}}{\partial \theta_k} = \frac{1}{N} \sum_{i=1}^{N} (y^{(i)} - f(x^{(i)}; \theta)) \frac{\partial f(x^{(i)}; \theta)}{\partial \theta_k}$
  end for
  $\theta_k := \theta_k - \alpha \frac{\partial \mathcal{L}_{MSE}}{\partial \theta_k} \qquad \forall k \in \{1 : K\}$
until $ep \geq N_{epoch}$

Algorithm 2 Minibatch Gradient Descent

Input: Dataset $\mathcal{D} = \{x^{(i)}, y^{(i)}\}_{i=1:N}$, Step Size α, Max Epochs N_{epoch}, Batch Size B
Output: Parameters $\{\theta_k\}_{k=1:K}$

Initialize θ randomly with small real numbers
$ep := 0$
repeat
  $ep := ep + 1$
  Divide $\mathcal{D}$ into batches $\{B_j\}_{j=1:\frac{N}{B}}$ of size B each
  for $j \in \{1 : \frac{N}{B}\}$ **do**
    for $k \in \{1 : K\}$ **do**
      $\frac{\partial \mathcal{L}_{MSE}}{\partial \theta_k} = \frac{1}{B} \sum_{i \in B_j} (y^{(i)} - f(x^{(i)}; \theta)) \frac{\partial f(x^{(i)}; \theta)}{\partial \theta_k}$
    end for
    $\theta_k := \theta_k - \alpha \frac{\partial \mathcal{L}_{MSE}}{\partial \theta_k} \qquad \forall k \in \{1 : K\}$
  end for
until $ep \geq N_{epoch}$

3.3. Parallel Minibatch Gradient Descent

Even with minibatches, the gradient computation can still be a bottleneck for most training algorithms. To mitigate this problem, observe that having the gradient as a sum of partial gradients with respect to individual training examples opens up the possibility of parallelizing the gradient computation efficiently. This can be done by distributing training examples of a minibatch across many processors, letting them compute a partial gradient over their own training examples and then summing up these partial gradients to get the approximate minibatch gradient. We use this to parallelize the gradient computation as shown in algorithm 3.

3.4. Backpropagation

It can be seen that a major subroutine in algorithms 1, 2 and 3 is the computation of gradient of the loss function for a single training example.

Since the neural network is a very complicated function

Algorithm 3 Parallel Minibatch Gradient Descent

Input: Dataset $\mathcal{D} = \{x^{(i)}, y^{(i)}\}_{i=1:N}$, Step Size α, Max Epochs N_{epoch}, Batch Size B, Num Threads T
Output: Parameters $\{\theta_k\}_{k=1:K}$

Initialize θ randomly with small real numbers
$ep := 0$
repeat
  $ep := ep + 1$
  Divide $\mathcal{D}$ into batches $\{B_j\}_{j=1:\frac{N}{B}}$ of size B each
  for $j \in \{1 : \frac{N}{B}\}$ **do**
    Divide B_j into thread-batches $\{B_{jt}\}_{t=1:\frac{B}{T}}$ of size $\frac{B}{T}$
    Thread 't' is forked:
    for $k \in \{1 : K\}$ **do**
      $\frac{\partial \mathcal{L}_{MSE}}{\partial \theta_k}\big|_t := \sum_{i \in B_{jt}} (y^{(i)} - f(x^{(i)}; \theta)) \frac{\partial f(x^{(i)};\theta)}{\partial \theta_k}$
    end for
    Thread 't' joins back 'main'
    for $k \in \{1 : K\}$ **do**
      $\frac{\partial \mathcal{L}_{MSE}}{\partial \theta_k} := \frac{1}{B} \sum_t \frac{\partial \mathcal{L}_{MSE}}{\partial \theta_k}\big|_t$
    end for
    $\theta_k := \theta_k - \alpha \frac{\partial \mathcal{L}_{MSE}}{\partial \theta_k} \qquad \forall k \in \{1 : K\}$
  end for
until $ep \geq N_{epoch}$

comprising of many layered operations, the gradient of loss function is not trivial to compute. For fully connected feed-forward neural networks, first a forward pass is made to compute the value of neural network activations for all layers as shown in algorithm 4. Then the gradient is computed by starting from the output layer and backpropagating gradient information for previous layers using the derivative chain-rule.

This procedure results in the famous *backpropagation* algorithm to compute the gradient described briefly as follows: Let x be a training example with label y. Our feed-forward neural network has an input layer and $L-1$ fully connected layers with sigmoid activation function. The parameters θ are all the neural network parameters i.e. $\theta = [W_{\{2:L\}}, b_{\{2:L\}}]$. We will use the Mean-Squared Error loss function:

$$\mathcal{L}_{MSE}([W_{\{2:L\}}, b_{\{2:L\}}]) = \frac{1}{N} \sum_{i=1}^{N} (y - a_L(x))^2 \qquad (6)$$

where $a_L(x)$ is the activation of the output layer when the input to the neural network is x and the network parameters are $[W_{\{2:L\}}, b_{\{2:L\}}]$. The backpropagation algorithm first computes errors $\delta_l \in \mathbb{R}^{n_l}$ and then computes the derivative of loss function with respect to parameters as shown in algorithm 5. Note that $a \circ b$ is the elementwise product between vectors a and b.

Algorithm 4 Forward Pass

Input: Example x, parameters $[W_{\{2:L\}}, b_{\{2:L\}}]$
Output: $z_l(x), a_l(x) \qquad \forall l = 1 : L$

$z_1(x) := x, a_1(x) := x$
for $l = 2 : L$ **do**
  $z_l(x) = (W_l)^T a_{l-1}(x) + b_l$
  $a_l(x) = \sigma(z_l)$
end for

Algorithm 5 Backpropagation

Input: Example x, label y, parameters $[W_{\{2:L\}}, b_{\{2:L\}}]$
Output: Derivatives $\{\frac{\partial \mathcal{L}_{MSE}}{\partial b_l}\}_{l=2:L}, \{\frac{\partial \mathcal{L}_{MSE}}{\partial W_l}\}_{l=2:L}$

Compute $z_l(x), a_l(x)\ \forall l = 1 : L$ with a forward pass
$\delta_L := \frac{\partial \mathcal{L}_{MSE}}{\partial a_L} \circ \sigma'(z_L(x))$
for $l = L : 2$ **do**
  $\frac{\partial \mathcal{L}_{MSE}}{\partial b_l} := \delta_l$
  $\frac{\partial \mathcal{L}_{MSE}}{\partial W_l} := a_{l-1} \delta_l^T$
  $\delta_{l-1} := (W_l \delta_l) \circ \sigma'(z_{l-1}(x))$
end for

4. Implementing BackPropagation on GPU

Graphics Processing Units (GPUs) are massively parallel processors that were designed for efficient computer graphics rendering and image processing. In fact, GPUs are very effective when used to execute the graphics rendering pipeline which is a sequence of geometric transformations on small multidimensional vectors. For this reason, GPUs are suitable for executing very fast certain linear algebra operations including but not limited to vector-vector addition, matrix-vector and matrix-matrix multiplication. Moreover, because GPUs consist of many throughput oriented multiprocessors that follow the Single Instruction Multiple Data (SIMD) execution model, they can be very useful for applications that require processing and transformation operations on large dataset.

Training neural networks is a process that can be realized through a series of matrix-matrix multiplications and additions which are applied for a large number of training examples. Every neural network can be defined and operated on by following this abstraction. Back propagation can be implemented as a series of kernel executions that are combined to ultimately compute the change in the weight values caused by the corresponding training batch. In our implementation, a neural network is viewed as a collection of layers, each one consisting of 3 distinct matrices. These matrices store the outgoing weight values W_i, the incoming activation values A_i and the delta error values D_i computed by back propagation. The activation values of the input layer (i.e. A_0) are the actual training examples in the corresponding batch. The activation values and the delta error values are stored in

column-major order. In contrast the weights for each layer are stored in row-major order. The bias values are embedded in the last column of the weight matrix.

We decompose back-propagation into 5 steps each one implemented by distinct kernel. The first step, handles the feed forward activation by implementing a tiled matrix-matrix multiplication based on Eq. 7. Here we denote with f the preferred activation function which is usually the sigmoid function. Our implementation is designed around templates and supports user defined activation functions. The feed-forward step is implemented using the tiled matrix-matrix multiplication as it is described in (). All threads participate in loading the tiles of the input matrices into shared memory. The threads use registers to accumulate partial results. After parsing the complete matrix each thread applies the activation function on the resulting value before storing it into global memory. All read and write operations to global memory are coalesced and there are no bank conflicts because thread iterate over the secondary matrix dimension in each case. An additional step is required to initialize the thread registers with the bias values.

$$A_{i+1} = f\left(W_i \cdot A_i + b_i\right), \forall i \in [1, L-1] \tag{7}$$

$$\delta_o utput_k ernel D_L = Y - A_L \tag{8}$$

$$D_i = W_i^T \cdot D_i \circ d(W_{i-1} \cdot A_{i+1}) \tag{9}$$

$$W_i = W_i + \frac{n}{b} \cdot \sum_{j=1}^{b} D_{i+1}^j \cdot (A_i^j)^T \tag{10}$$

Following this previous step, the delta error values are computed for each layer using Eq. 9. This equation is computed using 3 kernels, one for computing the output layer delta and two more that compute the transpose matrix-matrix multiplication and the hadamard product of the derivative for the corresponding activation function. A variation of the tiled matrix-matrix multiplication kernel is used to perform these operations. For the first kernel, the indexing is changed to enable multiplication with the transpose of the weight matrix. Here accesses to global memory remain coalesced since we order the thread blocks vertically on the weight matrix. However, we incur few bank conflicts when accessing the data in column-major order from shared-memory . In the second case, we chose to re-compute the product of $W_i \cdot A_i$ and multiply it with the result from the previous operation. This is because are goal was to support arbitrary defined activation functions. In case the sigmoid function is used, the derivative can be replaced with $A_i \cdot (1.0 - A_i)$ which enables significant reduction of the required computation.

Finally, we use the computed activation and the next layer delta matrices (Eq. 10) to update the weights of the corresponding layer depending on the chosen learning rate and batch size. This operation can be considered as another variation of matrix-matrix multiplication where each column from D_{i+1} and A_i are multiplied to produced a single matrix. The number of resulting matrices are equal to the batch size and are accumulated in thread registers. The resulting summation is multiplied by $\frac{n}{b}$ and added to the weight values of the corresponding layer. For this operation, access to global memory is again coalesced. However, shared memory access incurs many bank conflicts which are proportional to the batch size. In order to improve the performance, the corresponding tiles are loaded transposed into shared memory. This incurs only a single bank conflict during loading and avoids many bank conflicts during the multiplication phase.

5. Experimental setup

In this section, we first describe the datasets used for our experiments and the corresponding classification tasks. We then delineate the networks used for each of these datasets and the procedure to determine the values of hyperparameters. This is followed by a description of the evaluation metric of the speedup. All the algorithms were implemented in C++. The reported results were obtained on a Ubuntu 14.04.4 LTS system with 32 cores, 128 GB RAM and a clock speed of 2.6 GHz.

5.1. Dataset description

We test the speedups achieved by the parallel implementation on two public datasets - MNIST and KDD Cup 1999. These datasets have been widely used by the machine learning community to compare different learning algorithms. Their description is as follows:

- **MNIST** - This is an image database of handwritten digits which is commonly used for training various image processing systems (Figure 4). The task here is to classify the digit in the image. This dataset was derived from NIST's datasets and was formed by mixing the samples from NIST. This mixing was done because the training and testing datasets in the original NIST dataset were obtained from two different groups of people (Census Bureau employees and high school students). This dataset consists of 60,000 training images and 10,000 testing images, each of size 28$\times$ 28. In our experiments, we further divided the training set into training and validation in 5:1 ratio in order to determine suitable values for the hyperparameters.
- **KDD Cup 1999** - This is a dataset which first appeared in the KDD Cup held in 1999. The dataset has about 4 million data points each of which has 42 features. The features are of two types - categorical and integer. The

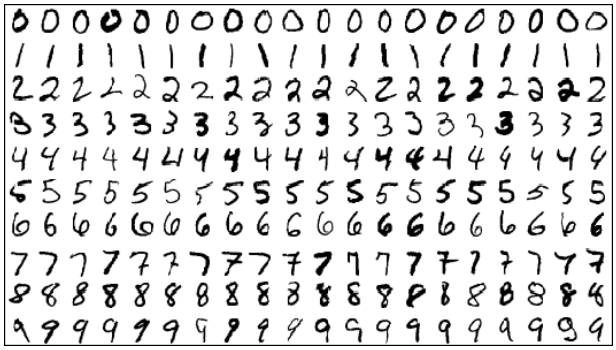

Figure 4. MNIST dataset

task in the competition was to build a network intrusion detector which can be used to distinguish between good and bad network connections.

5.2. Network details

We now describe the details of the structure of neural networks used. For our experiments, we use two different configurations of neural network -

- **Net-1h** - This network consists of 3 layers - input layer, one hidden layer and output layer. The input layer has 784 neurons for MNIST dataset and 42 neurons for KDD Cup 1999. The hidden layer is composed of 1024 neurons and the output layer has 10 neurons when used for MNIST dataset and 1 output for KDD Cup 1999.

- **Net-2h** - This network is composed of 4 layers - input layer, two hidden layers and output layer. The input and output layer are same as Net-1h and each of the hidden layers has 1024 neurons.

5.3. Hyperparameter selection

Hyperparameter selection, also known as model selection, is an important problem in machine learning. It refers to choosing the optimal hyperparameters for a learning algorithm. It is crucial to select the values for hyperparameters which generalize well. This enables the algorithm to perform well on test data.

In the context of neural networks, hyperparameters refer to learning rate and regularization constant. Since our implementation is free of regularization, we only have one hyperparameter - learning rate. To choose an optimal learning rate, we used the technique of grid search. As mentioned above, we divide the training set into training and validation. We use different learning rates defined in a grid and test the accuracy on the validation data. The rate which gives the highest accuracy on validation is chosen and used to evaluate the performance of the algorithm on test data.

5.4. Evaluation methodology

As has been previously mentioned, this work aims to speed up the learning of neural network by parallelizing backpropagation and running it on GPU. To evaluate our approach and quantify the usefulness of parallelization, we use speed up as our primary measure. We calculate this with respect to the serial implementation on the same network configuration. Speed up may sometimes be misleading and does not capture the efficiency of serial implementation. To resolve this we use another measure - GFLOPS (Billion Floating Point Operations Per Second). This measure is independent of serial implementation and can be used to supplement the speedup metric.

5.5. Experiments

In this paper, we implement the gradient descent algorithm for training neural network. The implementation is 3-fold - serial, parallelization with Pthreads and parallelization with CUDA. The implementation follows from section 3 and section 4. Please refer to these sections for a detailed explanation of the methods.

6. Results and Analysis

As described in previous sections, we compare our implementations with state of the art deep learning library, Theano, in three different parallelizations (Serial/Pthreads/Cuda-C). All experiments are done under the same network structures with MNIST datasets and the number of threads is fixed as 32.

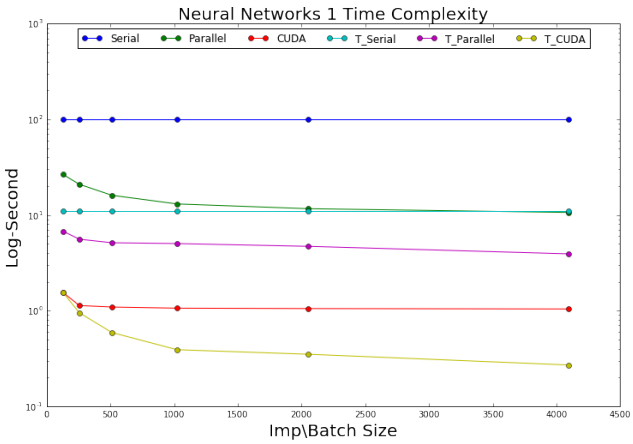

Figure 5. Learning time for Net-1h

Figure 5 clearly shows distinct characteristics between serial and parallel computation as well as CPU and GPU based computation. First, the training time for 1 epoch of the serial implementation (Blue curve and Light blue curve) is almost constant over the varying mini batch size. It is obvious result because regardless of the size of each mini batch training data is accessed sequentially. On the other hand, the parallel learning time (Green curve) shows decreased training

time due to multithreads distribution as we expected. One thing we should note is that the parallelism is enhanced when larger size of batch is used because the number of allocations of data into each thread is reduced. When smaller batch size is used, we need to assign each data into the corresponding thread more frequently and it causes more overheads. Moreover, as the last step, we need to combine all the results from every threads (which is called the reduction process) and the number of the reduction processes is also increased for smaller batch size. This behavior is shown commonly in the parallel computation based on Theano (T-Parallel).
In case of GPU based computation, the overall performance is significantly improved because GPU usually have much more cores which are specialized for repetitive operations such as matrix multiplications than CPU. As a result, GPU provides much faster learning time than that of CPU in general due to less number of allocation/reduction processes.
The faster computation by the parallelization is able to be captured by calculating GFLOPS. Figure 6 shows that parallelization provide much larger number of floating point operations than serial, thus, single thread computation.

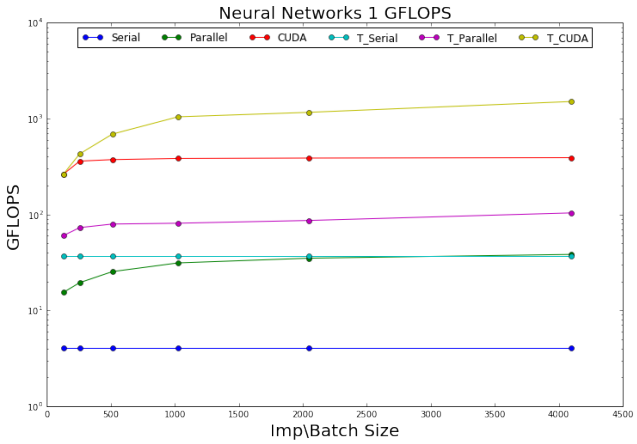

Figure 6. GFLOPS for Net-1h

BLAS As Basic Linear Algebra Subprograms, BLAS has been crucial workhorse in heavy numerical computing. By replacing the self-built matrix-vector product libraries with BLAS functions, we could obtain hugely improved results (even faster than Theano). Note that the execution time of BLAS doesn't depend on the number of threads because it parallelizes each Matrix-vector product and hence each thread executes different parts of the same example. The serial implementation is still a bit slower than Theano but is faster than our previous implementation by about 10 times. This way shows that the optimized parallelism gives speed-up over Naive serial implementation.

Bottleneck In ideal case, by increasing the size of the batch, the number of overheads (allocation/reduction) is exponentially decreased. However, our experiment results show that there are some performance bottlenecks. In our implementation, shared memory bank conflicts which reduce the parallelism when threads access the shared memory appear when the number of batch increases. When the dimensions (i.e.number of neurons between layers) of the neural network is not perfect multiple of 32 then some threads do not participate in the computation so they are doing no work but still have to wait on the barrier. As a result, it causes degraded parallelism.

Overall, we demonstrate that the parallel computation is significantly faster than the serial computation by utilizing multithreads based on the mini batch in the multicore machine. Moreover, by dividing training datasets into larger mini batches, we are able to better performance than that of smaller batch size due to reduced overheads. Similarly, the training time and GFLOPS are improved in the second networks (Net-2h) (See Figure 7,8).

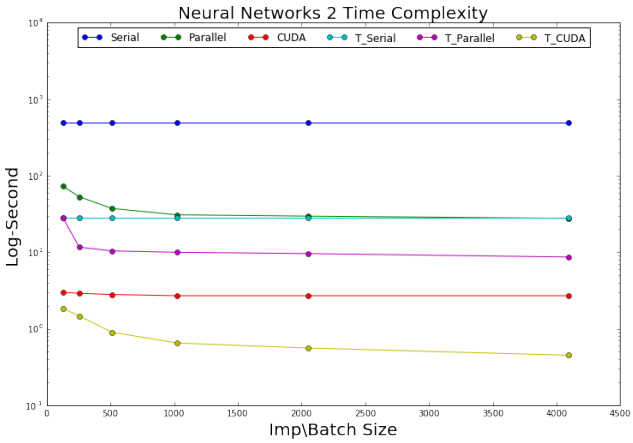

Figure 7. Learning time for Net-2h

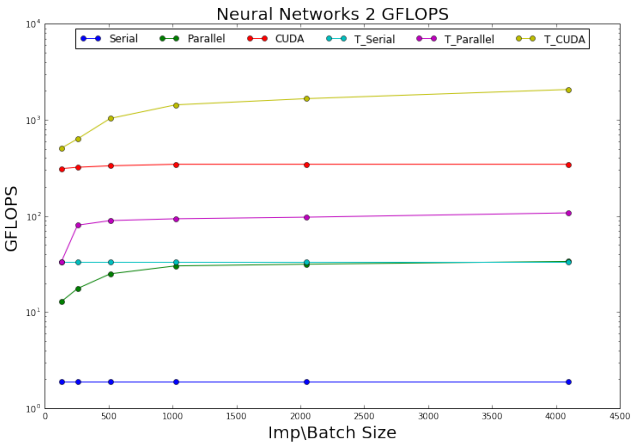

Figure 8. GFLOPS for Net-2h

7. Conclusion and Future Work

In this paper, we have explored a technique to parallelize training of multilayer feedforward neural networks by splitting the training examples among multiple threads of execution. We have obtained significant performance gains by parallelizing the training process both by using multiple cores and also on a GPU. We have demonstrated a -x times speedup using multiple cores only and then a -x times speedup using a GPU without any significant loss of accuracy. Our parallelized backpropagation implementation can find good usage for many real-time processing tasks. Typical examples include but are not limited to:

- Real-time online control of robotic manipulators which learn online from incoming sensory data.
- Self-driving cars which need to do online object recognition in real-time using deep neural networks.
- Speech translators which deploy deep recurrent neural networks for translation in real-time.