Parallel Gradient Descent for Multilayer Feedforward 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 in a serial implementation can be a very time consuming task. In this work, we have implemented parallel gradient descent to train multilayer feedforward neural networks. Specifically, we analyze two kinds of parallelization techniques: (a) parallel processing of multiple training examples across several threads and (b) parallelizing matrix operations for a single training example. We have implemented a serial minibatch gradient descent algorithm, its parallel multithreaded version (using Pthread library in C++), a BLAS parallelized version and a CUDA implementation on a GPU. All implementations have been compared and analyzed for the speedup obtained across various network architectures and increasing problem sizes. We have performed our tests on the benchmark dataset: MNIST, and finally also compared 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, fraud detection, image classification, diagnostic medicine applications and stock market prediction.

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

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 machines and/or across multiple machines.

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 involves computing the neural network output (forward propagation) and updating parameters by computing gradient values (backpropagation).

In this paper, we have implemented parallel minibatch gradient descent to train multilayer feedforward neural networks 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 gradient descent algorithm and its minibatch variant. It also describes the forward propagation and the backpropagation algorithms for neural networks. Section 4 discusses approaches to parallelization: (a) by distributing multiple examples across several threads, and (b) by performing matrix computations in parallel. It also discusses the implementation of parallel gradient descent on a GPU with CUDA. Section 5 describes our dataset and the experiments we have performed. Section 6 presents

our results obtained for these experiments 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 deep learning library Theano. Section 7 concludes the paper by discussing some potential applications and section 8 explores some potentially interesting future directions.

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 classification tasks.

2.1. Classification Problem

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

Neural networks are mathematical models partly inspired by the workings of the human brain. 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

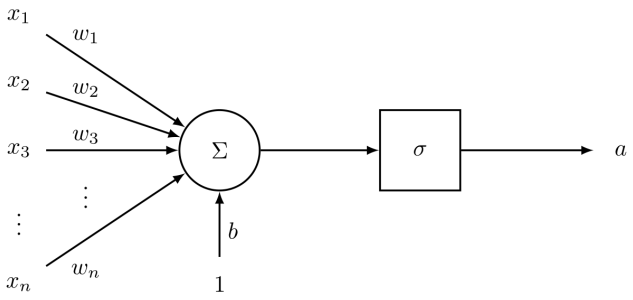

Figure 1. A neuron

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.

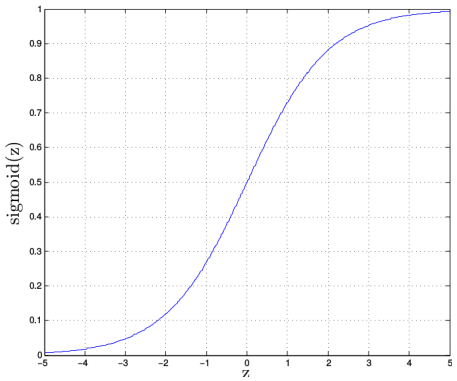

Figure 2. The sigmoid function (**?**)

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.

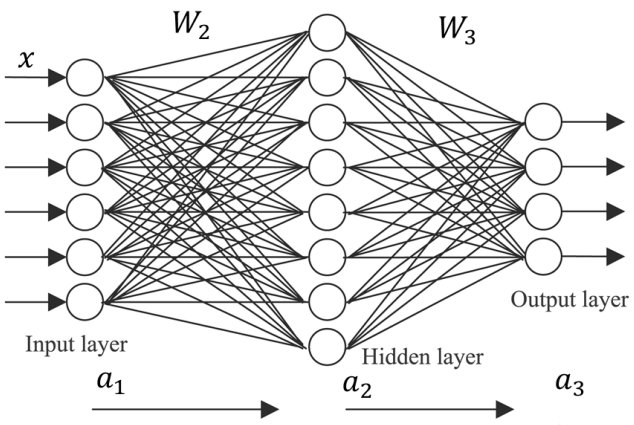

Figure 3. A neural network with one hidden layer

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 unchanged i.e. $a_1 = x_1 = x$ (linear activation function).

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 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.

An example network with one hidden layer is shown in figure 3.

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. A naive implementation of Gradient Descent is as shown in algorithm 1.

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
  $\theta_k := \theta_k - \alpha \frac{\partial \mathcal{L}_{MSE}}{\partial \theta_k} \quad \forall k \in \{1 : K\}$
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
    $\theta_k := \theta_k - \alpha \frac{\partial \mathcal{L}_{MSE}}{\partial \theta_k} \quad \forall k \in \{1 : K\}$
  end for
until $ep \geq N_{epoch}$

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.

3.3. Backpropagation

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

Since the neural network is a very complicated function comprising of many layered operations, the gradient of loss function is not trivial to compute. For fully connected feed-

Algorithm 3 Forward Propagation

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

$z_1(x) := x, a_1(x) := x$
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 4 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

forward neural networks, first a forward pass is made to compute the value of neural network activations for all layers as shown in algorithm 3. 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 4 (**?**). Note that $a \circ b$ is the elementwise product between vectors a and b.

4. Parallel Gradient Descent

Even with minibatches, the gradient computation can still be a bottleneck for most training algorithms. There are two potential ways to get rid of this problem and we describe them in the subsequent subsections.

Algorithm 5 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 $ep \geq N_{epoch}$

4.1. Parallel Minibatch Gradient Descent

Observe that the gradient is a sum of partial gradients with respect to the individual training examples. So one way to parallelize gradient computation could be by distributing training examples of a minibatch across many processes, letting them compute a partial gradient over their own training examples and then summing up these partial gradients to get the approximate minibatch gradient. This approach results in a procedure shown in algorithm 5.

4.2. Parallelizing Matrix Computations

An alternative way to parallelize the gradient computation process can be by parallelizing individual steps of backpropagation (algorithms 3 and 4). Note that the forward propagation and backpropagation algorithms comprise various matrix vector multiplications which can be individually parallelized across multiple threads of execution. Parallelized matrix-vector multiplications are already efficiently implemented in various linear algebra libraries and we will use the BLAS library to implement this second method of parallelization.

4.3. Parallelizing on GPUs

Graphics Processing Units (GPUs) are massively parallel processors for efficiently executing certain 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 large amount of data.

Backpropagation can be accelerated on a GPU using a series of matrix-matrix multiplication and addition kernels. Our implementation relies on several variations of the tiled matrix-matrix multiplication kernel which is available in CUDA samples. The original problem is decomposed in 5 kernel executions per training example which include the computation of the activation values, the calculation of the output error delta and the hidden layer delta values, the calculation of the derivative activation values and the final summation of weights. In the case of computing the delta values, we included an optimization when the sigmoid function is used which avoids recomputing the activation values by deriving the derivative of the activation function from the values of the feed forward step. Also, when computing the final ΔW that is used to update the weights for each layer, our initial kernel was incurring many shared memory bank conflicts during computation. To improve performance, we chose to invert the order in which tiles are stored in shared memory. This way only a single bank conflict occurs when loading data into shared memory while there are zero bank conflicts during the computation step.

5. Experimental setup

In this section, we first describe the dataset used for our experiments and the corresponding classification task. 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. The GPU used was Nvidia Tesla K40C.

5.1. Dataset description

We test the speedups achieved by the parallel implementation on the benchmark dataset: MNIST (**?**). It has been widely used by the machine learning community to compare different learning algorithms.

MNIST is an image database of handwritten digits, commonly used for training various image processing systems (Figure 4). The task is to classify the digit $(0-9)$ 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). The dataset consists of 60,000 training images and 10,000 testing images, each of size 28×28. 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.

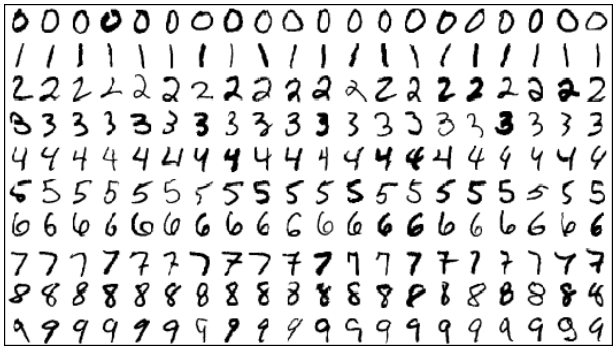

Figure 4. MNIST dataset

5.2. Network details

We used two different configurations of neural networks:

- **Net-1h**: 3 layered network - input (784), one hidden layer (1024) and output layer (10).
- **Net-2h**: 4 layered network - input (784), two hidden layers (1024 each) and output layer (10).

Both networks had linear activations for input layer and sigmoid activations for all subsequent layers. The dense layer for each network was initialized using glorot uniform distribution (**?**) and the biases were initialized with zeros.

5.3. Hyperparameter selection

Hyperparameter selection (model selection) is an important problem in machine learning which involves choosing optimal values for hyperparameters that generalize well on test data.

We chose our learning rate hyperparameter by doing a grid search on the grid $[0.001, 0.002, 0.01, 0.1, 0.5, 1.0]$. As mentioned above, we divide the training set into training and validation (5:1 split). We use different learning rates defined in the grid for learning and test the accuracy on 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. Experiments

In this paper, we implement different versions of gradient descent algorithm for training neural networks. Some of them used our self-written (slow) linear algebra library (called LINALGLIB henceforth), and others used BLAS. All parallel multicore implementations used 32 execution threads. Keras (**?**) and Theano (**??**) based implementations also use

BLAS. The implementations are as follows:

- Naive Serial, Multicore Parallel, GPU Parallel
- Fast Serial, Fast Parallel
- Theano Serial, Theano Parallel, Theano GPU

Detailed description of implementations is given in table 1.

Name	Parallelism	Lin. Algebra Library	Platform
Naive Serial	-	LINALGLIB	C++
Multicore Parallel	Training examples	LINALGLIB	Pthreads, C++
GPU Parallel	Training examples	LINALGLIB	CUDA, C++
Fast Serial	-	BLAS	C++
Fast Parallel	Matrix operations	BLAS	C++
Theano Serial	-	BLAS	Keras, Theano
Theano Parallel	Training ex., Matrix ops	BLAS	Keras, Theano
Theano GPU	Training ex., Matrix ops	BLAS	Keras, Theano

Table 1. Detailed description of implementations

5.5. Evaluation methodology

To evaluate our approach and quantify the usefulness of parallelization, we use speedup as our primary measure. We compute speedup for each parallel implementation with respect to its serial counterpart on the same network configuration. For each parallel implementation we also compute the running time per training epoch (in seconds) and the amount of computation per epoch (in gigaflops) since speedup alone may sometimes be misleading and does not capture the efficiency of parallel implementation. These measures are independent of serial implementation and can be used to supplement the speedup metric.

6. Results and Analysis

All our networks give us about 93% to 97.8% classification accuracy on validation and test sets, but in this work we did not focus on getting a better classification accuracy. Instead our major focus was to analyze the speedup obtained by parallelization, and the gigaflops of computation obtained for different batch sizes.

Figures 5 and 7 show bar plots of the time per epoch (in seconds, called TPE henceforth) by all implementations as the batch size increases. Figures 6 and 8 show the corresponding bar plots for amount of computation performed

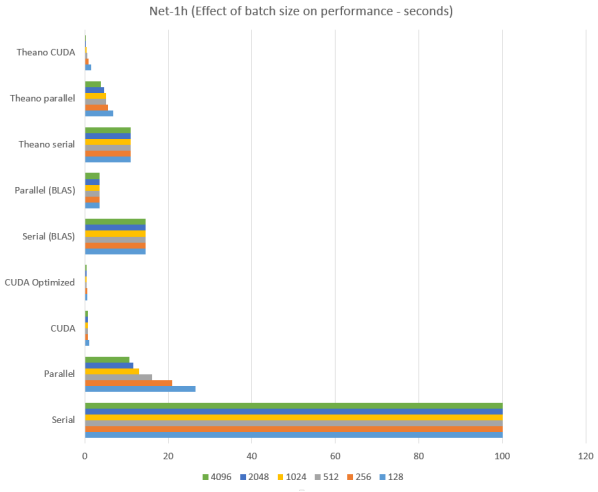

Figure 5. Time per epoch for Net-1h

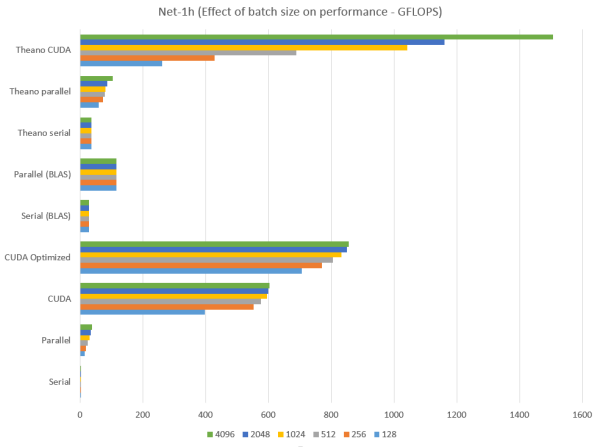

Figure 6. GFLOPS for Net-1h

per epoch (in gigaflops, called GFLOPS henceforth). The speedups (called SUP henceforth) obtained for Net-1h and Net-2h are shown in figures 9 and 10 respectively for various batch sizes.

The following points can be immediately observed from the figures:

- TPE and GFLOPS are constant for serial implementations regardless of batch sizes, which is to be expected because serial implementations access all data sequentially regardless of batch size.
- TPE decreases and GFLOPS increases with increase in batch size for parallel and GPU implementations, which parallelize on training examples. This is expected since having more training examples in a batch reduce the thread creation overheads.
- GPUs have a huge number of parallel cores and threads per core and hence they clearly dominate over the mul-

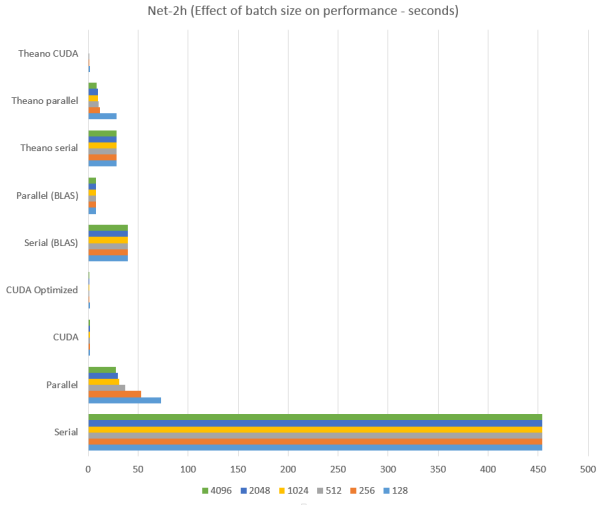

Figure 7. Time per epoch for Net-2h

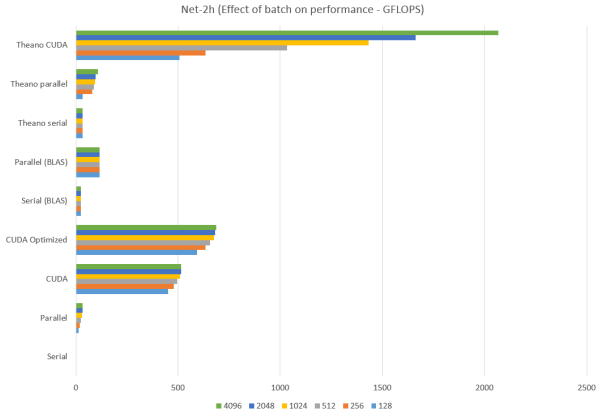

Figure 8. GFLOPS for Net-2h

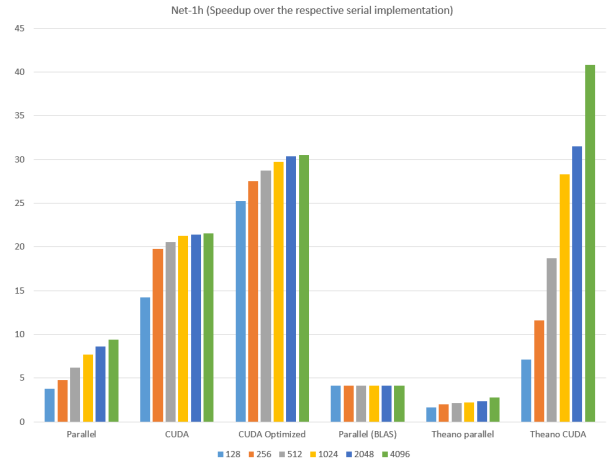

Figure 9. Speedup for Net-1h

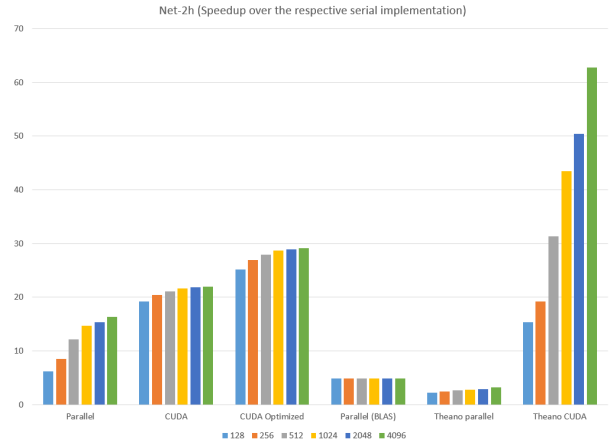

Figure 10. Speedup for Net-2h

ticore parallel implementations at all values of batch sizes and network configurations.

- BLAS parallel implementations do not exhibit a change in TPE or GFLOPS with increasing batch sizes, since they do not parallelize on minibatches.
- Multicore Parallel and CUDA GPU implementations give an average speedup of approximately 10 and 25 respectively.
- Theano serial and BLAS serial implementations are very efficient already and hence their parallel counterparts demonstrate a smaller speedup, although the parallel counterparts parallelizing on matrix computations clearly win over the parallel implementations parallelizing on training examples, in terms of GFLOPS.
- Theano implementations (serial, parallel and GPU) use both types of parallelization and hence demonstrate lower TPE than all their corresponding counterparts for large batch sizes where efficiency of parallelized matrix operations matters a lot.
- For smaller batch sizes around 128, our CUDA implementation dominates over Theano GPU because parallelization of matrix computations have a larger overhead at such scales with smaller matrix sizes. This is also reflected in Theano GPU's speedup, which grows extremely fast as batch size grows (figures 9 and 10).

BLAS As Basic Linear Algebra Subprograms, BLAS has been a crucial workhorse in heavy numerical computing. By replacing our self-built LINALGLIB with BLAS, we obtained hugely improved results. 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.

GPU Bottlenecks Ideally the number of overheads (allocation/reduction) should be monotonically reduced by increasing the size of the batch. However, our experiment results show that there are some performance bottlenecks. In our implementation, shared memory bank conflicts which re-

duce the parallelism when threads access the shared memory appear when the number of batches increase. When the dimensions (i.e. number of neurons between layers) of the neural network are not perfect multiples of 32 then some threads do not participate in the computation which results in degraded parallelism.

Overall, we demonstrate that the parallel computation is significantly faster than the serial computation if we utilize multiple threads for processing many examples or do matrix-vector computations in parallel. By dividing training datasets into larger mini batches, we are able to perform better due to reduced parallelization overheads.

7. Conclusion

In this paper, we have explored techniques to parallelize training of multilayer feedforward neural networks by (a) splitting training examples among multiple threads of execution, and (b) by parallelizing matrix computations for a single training example. 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 $\times 10$ speedup with multiple cores and $\times 25$ on a GPU by parallelizing across multiple training examples, and a heavy increase in GFLOPS by parallelizing matrix computations for a single training example. Our parallelized gradient descent 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.

8. Future Work

It would be an interesting idea to combine the two parallelization techniques that we have explored in this paper as future work. Specifically, it might be possible to speed up the training even more by splitting training examples in parallel, and then further hierarchically parallelizing matrix computations for each individual example.

References

Bastien, Frédéric, Lamblin, Pascal, Pascanu, Razvan, Bergstra, James, Goodfellow, Ian J., Bergeron, Arnaud, Bouchard, Nicolas, and Bengio, Yoshua. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

Baydin, Atlm Gne. Neural networks. URL http://diffsharp.github.io/DiffSharp/img/examples-neuralnetworks-neuron.png.

Bergstra, James, Breuleux, Olivier, Bastien, Frédéric, Lamblin, Pascal, Pascanu, Razvan, Desjardins, Guillaume, Turian, Joseph, Warde-Farley, David, and Bengio, Yoshua. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, june 2010.

Chollet, Franois. keras. https://github.com/fchollet/keras, 2016.

Glorot, Xavier and Bengio, Yoshua. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pp. 249–256, 2010.

Lecun, Yann and Cortes, Corinna. The mnist database of handwritten digits. URL http://yann.lecun.com/exdb/mnist/.

Nielson, Michael. *Neural Networks and Deep Learning*. URL http://neuralnetworksanddeeplearning.com/index.html.