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 task, 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 same implementations done with a 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 describe the task of training feedforward neural networks for classification tasks using the conventional backpropagation algorithm. Feedforward neural networks act as function approximators in such tasks.

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 weights (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. Now we describe a feedforward neural network.

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

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

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 are 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. The sigmoid function is defined as follows:

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

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, $N - 2$ hidden layers and a final output layer. The l^{th} layer has N_l neurons ($N = \sum_{l=1}^{L} N_l$) The first layer is the input layer and contains dummy neurons

Add details specific to neural nets

3. Serial and Parallel Backpropagation

which in its most naive form is implemented as follows:

1. Initialize all weights (θ) randomly with small values close to 0.
2. Repeat until convergence {

$$\theta_k := \theta_k - \alpha\frac{\partial \mathcal{L}_{MSE}}{\partial \theta_k} \qquad \forall k \in \{1, 2, ..., K\}$$

}

Note that the derivative of the loss function generally takes the following form:

$$\frac{\partial \mathcal{L}_{MSE}}{\partial \theta_k} = \frac{1}{N}\sum_{i=1}^{N}(y_i - f(x_i;\theta))\frac{\partial f(x_i;\theta)}{\partial \theta_k}$$

Since this requires a summation over all the training examples, the gradient computation is the biggest bottleneck for many supervised learning tasks on huge data sets.

On the other hand, 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 across various processors/machines and letting them compute a partial gradient over their own training examples and then summing up these partial gradients to get the actual gradient. A good approximation of this process can be obtained by using a mini-batch of training examples per iteration, instead of using the full dataset.

Describe backpropagation serial implementation in detail

Describe platform used
Describe backpropagation parallelization using pthreads

4. Parallel Backpropagation on a GPU

Describe platform
Describe CUDA implementation for GPU

5. Experiments

Describe datasets and classification task
Decribe experimental network sizes
Describe hyperparameter selection
Describe experiments: serial vs pthreads vs cuda vs theano.

- Serial implementation of Mini-batch Gradient Descent(MGD) in C++
- Parallel implementation of MGD in C++ using Pthreads
- Comparison between serial MGD, parallel MGD with OpenMP and Theano's gradient descent (CPU)
- Parallel MGD using Cuda-C on a GPU
- Comparison between serial MGD, parallel MGD (Cuda) and Theano's gradient descent (GPU)

To test the above implementations, we will use the following datasets:

- **MNIST** - This is a database of handwritten digits that is commonly used for training various image processing systems. It has 60,000 images each of size 28×28 pixels. The task is to classify the digit in the image.
- **KDD Cup 1999** - This is a data set designed for intrusion detection (distinguishing bad network connections from good ones). It has about 4 million data points each of which has 42 features.

Note that since Theano is a state of the art Deep Learning library which performs many other optimizations apart from just parallelizing Neural Network training, we do not expect our code to run faster than Theano. Instead the comparison will be done to see how close we can get to the current state of the art by simply parallelizing the Neural Network training.

6. Results and Analysis

Provide experiment results: serial
Compare serial vs pthreads vs cuda vs theano.
Provide graphs for speedup with increasing problem size
Analyze speedup with number of threads/cores used
Analyze the level of parallelization obtained
Analyze performance bottlenecks
Do overall analysis and compare obtained results with your expectations

7. Future Work

Discuss application of parallel training and future usage of this work