Parallel Implementation of Deep Learning Using MPI

CSE633 Parallel Algorithms (Spring 2014)
Instructor: Prof. Russ Miller
Team #13: Tianle Ma
Email: tianlema@buffalo.edu

May 7, 2014
Introduction to Deep Belief Network

Parallel Implementation Using MPI

Experiment Results and Analysis
Why Deep Learning?

• Deep Learning is a set of algorithms in machine learning that attempt to model high-level abstractions in data by using architectures composed of multiple non-linear transformations.

• It has been the hottest topic in speech recognition, computer vision, natural language processing, applied mathematics, ... in the last 2 years

• Deep Learning is about representing high-dimensional data

• It's deep if it has more than one stage of non-linear feature transformation
Deep Belief Network

- Probabilistic Generative model.
- Contains multiple layers of nonlinear representation.
- Fast, greedy layer-wise pretraining algorithm.
- Inferring the states of the latent variables in highest layers is easy.
Deep Belief Network

Internal representations capture higher-order statistical structure

Higher-level features: Combination of edges

Low-level features: Edges

Built from *unlabeled* inputs.

Input: Pixels

(Hinton et. al. Neural Computation 2006)
Deep Belief Network

Stacked Restricted Boltzmann Machines

Approximate Inference

\[ Q(h^3|h^2) \]
\[ Q(h^2|h^1) \]
\[ Q(h^1|v) \]

Generative Process

\[ P(h^2, h^3) \]
\[ P(h^1|h^2) \]
\[ P(v|h^1) \]

\[ Q(h^t|h^{t-1}) = \prod_j \sigma \left( \sum_i W^t h_i^{t-1} \right) \]
\[ P(h^{t-1}|h^t) = \prod_j \sigma \left( \sum_i W^t h_i^t \right) \]
Restricted Boltzmann Machines

$$p(v, h) = \frac{1}{\mathcal{Z}} e^{h^T W v + b^T v + a^T h}$$

**Restricted:** No interaction between hidden variables

Inferring the distribution over the hidden variables is easy:

$$P(h \mid v) = \prod_j P(h_j \mid v) \quad P(h_j = 1 \mid v) = \frac{1}{1 + \exp(-\sum_i W_{ij} v_i - a_j)}$$

**Factorizes:** Easy to compute

Similarly:

$$P(v \mid h) = \prod_i P(v_i \mid h) \quad P(v_i = 1 \mid h) = \frac{1}{1 + \exp(-\sum_j W_{ij} h_j - b_i)}$$

Markov random fields, Boltzmann machines, log-linear models.
Approximate ML Learning for RBMs

Run Markov chain (alternating Gibbs Sampling):

\[
P(h|v) = \prod_j P(h_j|v) \quad P(h_j = 1|v) = \frac{1}{1 + \exp(-\sum_i W_{ij}v_i - a_j)}
\]

\[
P(v|h) = \prod_i P(v_i|h) \quad P(v_i = 1|h) = \frac{1}{1 + \exp(-\sum_j W_{ij}h_j - b_i)}
\]
Contrastive Divergence

A quick way to learn RBM:

\[ P(h|v) \]
\[ h \quad \quad \quad \quad v \]

\[ P(v|h) \]

Data \quad Reconstructed Data

- Start with a training vector on the visible units.
- Update all the hidden units in parallel.
- Update the all the visible units in parallel to get a “reconstruction”.
- Update the hidden units again.

Update model parameters:

\[ \Delta W_{ij} = E_{P_{\text{data}}}[v_i h_j] - E_{P_1}[v_i h_j] \]
Algorithm 1  RBMupdate \((v_0, \epsilon, W, b, c)\)

- **for all** hidden units \(i\) do
  
  Compute \(Q(h_{0i}|v_0) = \text{sigm}(b_i + \sum_j W_{ij}v_{0j})\) (for binomial units)

  Sample \(h_{0i}\) from \(Q(h_{0i}|v_0)\)

- **end for**

- **for all** hidden units \(j\) do
  
  Compute \(P(v_{1j}|h_0) = \text{sigm}(c_j + \sum_i W_{ij}h_{0i})\) (for binomial units)

  Sample \(v_{1j}\) from \(P(v_{1j}|h_0)\)

- **end for**

- **for all** hidden units \(i\) do
  
  Compute \(Q(h_{0i}|v_0) = \text{sigm}(b_i + \sum_j W_{ij}v_{0j})\) (for binomial units)

  Sample \(h_{0i}\) from \(Q(h_{0i}|v_0)\)

- **end for**
**Algorithm 1** RBM update \((v_0, \epsilon, W, b, c)\)

- \(W \leftarrow W - \epsilon(h_0 v'_0 - Q(h_1 = 1|v_1) v'_1)\)
- \(b \leftarrow b - \epsilon(h_0 - Q(h_1 = 1|v_1))\)
- \(c \leftarrow c - \epsilon(v_0 - v_1)\)

Update model parameters

Feature representation

Contrastive Divergence

Gibbs Step
Algorithm2 PreTrainDBN \((x, \epsilon, L, n, W, b)\)

- Initialize \(b^0 = 0\)
- for \(l = 1\) to \(L\) do
  - Initialize \(W^l = 0, b^l = 0\)
  - while not stopping criterion do
    - \(g^0 = x\)
    - for \(i = 1\) to \(l - 1\) do
      - Sample \(g^i\) from \(Q(g^i|g^{i-1})\)
    - end for
    - RBMupdate \((g^{l-1}, \epsilon, W^l, b^l, b^{l-1})\)
  - end while
- end for

Stacked RBMs!

Unsupervised Learning

Learn Latent Variables
(Higher level Feature Representations)
**Algorithm 3** FineTuneDBN \((x, y, \epsilon, L, n, W, b)\)

- \(\mu^0(x) = x\)
- **for** \(l = 1 \text{ to } L\) **do**
  
  \[
  \mu^l(x) = \mathbb{E}[g^i | g^{i-1} = \mu^{l-1}(x)] = \text{sigm}(b^l_j + \sum_k W^l_{jk} \mu^l_k(x)) \text{ (for binomial units)}
  \]

- **end for**

- Network output function: \(f(x) = V(\mu^l(x)', 1)'\)

- Use Stochastic Gradient Descent to iteratively minimize cost function \(C(f(x), y)\) (Back Propagation)

---

**Supervised Learning**  
Fine tune Model parameters
Learning Deep Belief Network

Step1:
Unsupervised generative pre-training of stacked RBMs (Greedy layer wise training)

Step2:
Supervised fine-tuning (Back Propagation)

How many parameters to learn?

$$\sum_{i=0}^{L} n_i n_{i+1} + \sum_{i=0}^{L+1} n_i$$
Can We SCALE UP?

• Deep learning methods have higher capacity and have the potential to model data better.

• More features always improve performance unless data is scarce.

• Given lots of data and lots of machines, can we scale up deep learning methods?

  MapReduce? No!

  MPI? Yes!
For large models, partition the model across several machines.

Models with local connectivity structures tend to be more amenable to extensive distribution than fully-connected structures, given their lower communication costs.

Need to manage communication, synchronization, and data transfer between machines.

Reference Implementation: Google DistBelief
Data Parallelism (Asynchronous SGD)

Before processing each batch, a model replica asks the Parameter Server for an updated copy of its model parameters;

- Compute a parameter gradient.
- Send the parameter gradient to the server. Parameter Server applies the gradient to the current value of the model parameters.

Divide the data into a number of subsets and run a copy of the model on each of the subsets.
Fetch parameters $w$ \\
Push gradients $\Delta w$
Algorithm 4: Asynchronous SGD(\(\alpha\))

- **Procedure** `StartFetchingParameters(parameters)`
  \(\text{parameters} \leftarrow \text{GetParametersFromParameterSever}()\)

- **Procedure** `StartPushingGradients(gradients)`
  \(\text{SendGradientsToParameterSever}()\)

- **Main**
  
  **Global** `parameters, gradients`

  ```
  while not stopping criterion do
    \text{StartFetchingParameters}() \hspace{1cm} \text{MPI\_Get}() \hspace{1cm} \text{Train DBN} \hspace{1cm} \text{Takes lots of time}
    \text{data} \leftarrow \text{GetNextMiniBatch}() \hspace{1cm} \text{Update the parameters}
    \text{gradients} \leftarrow \text{ComputeGradient}() \hspace{1cm} \text{Update the parameters}
    \text{parameters} \leftarrow \text{parameters} + \alpha \times \text{gradients}
    \text{StartPushingGradients}() \hspace{1cm} \text{Update the parameters}
  end while
  ```
Experiment

- MNIST Handwritten Dataset
  - $28 \times 28 = 784$ pixels
  - 60000 training images
  - 10000 test images

- Partition the training data into data shards for each model replica for parallelism.
Cores VS Time (#Iterations = 100)

- Equally divide the training data into #Total Cores partitions (Balanced Partitions).

- The smaller training data, the less training time which is dominate in the total time.

- After about 10 partitions, the training data is small enough, the training time is not dominate in the total time, so the speed-up is not increasing linearly.
Cores VS Speed-up (#Iterations = 100)

- Equally divide the training data into #Total Cores partitions (Balanced Partitions).
- The smaller training data, the less training time which is dominate in the total time.
- After about 10 partitions, the training data is small enough, the training time is not dominate in the total time, so the speed-up is not increasing linearly.
Fixing \#Node\(=\) 2, Accuracy > 90%

- Time begins increasing after about 20 data partitions.
- Reason:
  - Data partition becomes too small and insufficient to learn the model parameters.
  - So it needs more iterations to get the same accuracy.
Fixing #Node= 2, Accuracy > 90%

- Speed-up begins decreasing after about 20 data partitions.
- Reason:
  - Data partition becomes too small and insufficient to learn the model parameters.
  - So it needs more iterations to get the same accuracy.
Fixing #Tasks Per Node (TPN) = 2

- Time begins increasing after about 20 data partitions.
- Reason:
  - Data partition becomes too small and insufficient to learn the model parameters.
  - So it needs more iterations to get the same accuracy.

Similar to the results of fixing #Nodes
Fixing #Tasks Per Node (TPN) = 2

- Speed-up begins decreasing after about 20 data partitions.
- Reason:
  - Data partition becomes too small and insufficient to learn the model parameters.
  - So it needs more iterations to get the same accuracy.

Similar to the results of fixing #Nodes, slightly different.
Time begins increasing after about 20 data partitions.

Reason:
- Data partition becomes too small and insufficient to learn the model parameters.
- So it needs more iterations to get the same accuracy.

Inter-communication cost is higher than intra-communication cost.
Speed begins decreasing after about 20 data partitions.

Reason:
- Data partition becomes too small and insufficient to learn the model parameters.
- So it needs more iterations to get the same accuracy.

Inter-communication cost is higher than intra-communication cost.
## Results

### Table 1: Fixing Nodes

<table>
<thead>
<tr>
<th>#Node</th>
<th>#TPN</th>
<th>#Total Cores</th>
<th>Time/s</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1413</td>
<td>3.1552</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>674</td>
<td>6.6152</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>12</td>
<td>405</td>
<td>11.0152</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>16</td>
<td>314</td>
<td>14.2152</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>20</td>
<td>318</td>
<td>14.0152</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>24</td>
<td>342</td>
<td>13.0152</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>28</td>
<td>428</td>
<td>10.4152</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>32</td>
<td>741</td>
<td>6.0152</td>
</tr>
</tbody>
</table>

### Table 2: Fixing Tasks Per Nodes (TPN)

<table>
<thead>
<tr>
<th>#Node</th>
<th>#TPN</th>
<th>#Total Cores</th>
<th>Time/s</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1411</td>
<td>3.1552</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>8</td>
<td>728</td>
<td>6.1254</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>12</td>
<td>438</td>
<td>10.1854</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>16</td>
<td>337</td>
<td>13.2054</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>20</td>
<td>337</td>
<td>13.2054</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>24</td>
<td>365</td>
<td>12.2054</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>28</td>
<td>474</td>
<td>9.4054</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>32</td>
<td>856</td>
<td>5.2054</td>
</tr>
</tbody>
</table>

Inter-communication costs between nodes are higher than intra-communication costs between nodes.
Conclusion

- There is a tradeoff between communication costs and computation costs. Inter-communication costs > Intra-communication costs.

- When each data partition is big, the training time of DBN dominates. The speed-up on CCR using MPI is approximately linear.

- When the partition becomes small enough, it’s insufficient to train sophisticated DBN model. To achieve certain accuracy, it needs more iterations. The performance could become significant worse when the partition is too small. It depends on the datasets. The bigger dataset, the more amenable to extensive distribution and the more obvious speed-up.

- In general, using MPI framework to distribute large deep neural network is a good choice. The efficiency and scalability have been proved in industrial practice.
Reference


• Jeffrey Dean, Greg S. Corrado, etc. *Large Scale Distributed Deep Networks*. NIPS, 2012


• [http://deeplearning.net/tutorial/](http://deeplearning.net/tutorial/)
Thank You for Your Time