Fast Tree-Structured Recursive Neural Tensor Networks

Anand Avati, Nai-Chia Chen

Stanford University
avati@cs.stanford.edu, ncchen@stanford.edu
Project TA: Youssef Ahres

1 Introduction

In this project we explore different ways in which we can optimize the computation of training a Tree-structured RNTN, in particular batching techniques in combining many matrix-vector multiplications into matrix-matrix multiplications, and many tensor-vector operations into tensor-matrix operations. We assume that training is performed using mini-batch AdaGrad algorithm, and explore how we can exploit the presence of multiple examples and batch computation across the set as a whole. We explore how we can apply our optimization techniques to the forward propagation phase, the back propagation phase, and run the batched operations on GPUs. Our goal is to speed up the execution of Tree-structured RNNs so that its runtime performance is no more a limiting factor for adoption.

We use the Stanford CoreNLP project that has an implementation of RNTN in Java as our baseline. All our implementation and experiments are performed over this.

2 Background - Recursive Neural Tensor Networks

Recursive Neural Tensor Network (RNTN) is a model for semantic compositionality, proposed by Socher et al [1]. This network has been successfully applied to sentiment analysis, where the input is a sentence in its parse tree structure, and the output is the classification for the input sentence, i.e., whether the meaning is very negative, negative, neutral, positive, or very positive.

2.1 Forward propagation

In the forward phase, each input is a sentence in its parse tree structure. (See figure (1)). Each word in the input sentence is converted to a d-dimensional word vector through a word embedding matrix $L \in \mathbb{R}^{d \times |V|}$, where |V| is the size of the vocabulary. Further more, each word vector is converted to a d-dimensional node vector through the element-wise t function and stored in the leaf node of the tree. The node vectors of internal nodes are then computed in a bottom-up fashion as follows. Let x^i be the parent node of its left and right children nodes x_L^i , x_R^i , then the node vector at x^i is defined as

$$x^{i} = \tanh\left(\begin{bmatrix} x_{L}^{i} \\ x_{R}^{i} \end{bmatrix}^{T} V \begin{bmatrix} x_{L}^{i} \\ x_{R}^{i} \end{bmatrix} + W \begin{bmatrix} x_{L}^{i} \\ x_{R}^{i} \end{bmatrix}\right) \in \mathbb{R}^{d}, \tag{1}$$

where $W \in \mathbb{R}^{2d \times d}$ is the weight matrix, $V \in \mathbb{R}^{2d \times 2d \times d}$ is the weight tensor, and tanh applies on vectors element-wise.

Each node in the tree is now a d-dimensional vector. The network predicts the meaning (very negative, negative, neutral, positive, or very positive) of every node by producing a probability vector $y^i \in \mathbb{R}^5$ defined as

$$y^i = \operatorname{softmax}(W_s x^i) \in \mathbb{R}^5, \tag{2}$$

where $W_s \in \mathbb{R}^{5 \times d}$ is the sentiment classification matrix. Moreover, each node x^i is associated with a ground truth (or target vector) $t^i \in \mathbb{R}^5$, which is a binary vector whose j-th component is 1 if j is the correct label and is 0 in all other components. In order to minimize the KL-divergence between the predicted distribution y^i and the true distribution t^i , the error function with regularization is defined as

$$E(\theta) = -\sum_{i} \langle t^{i}, \log y^{i} \rangle + \lambda ||\theta||^{2},$$

where the model parameters are $\theta = (L, W, W_s, V)$, and the log function applies on y^i elementwise.

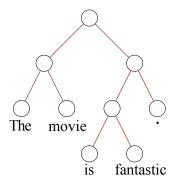


Figure 1: Each training example is a sentence.

2.2 Backpropagation [2]

The formulae for errors at the node x^i are as follows. Let $\delta^{i,s}$ denote the softmax error and $\delta^{i,com}$ denote the complete incoming error, then

$$\begin{split} \delta^{i,s} &= W_s^T(y^i - t^i), \\ \delta^{i,com} &= \begin{cases} \delta^{i,s}, & \text{if } x^i \text{ is the root} \\ \delta^{i,s} + \delta^{p(i),down}[1:d], & \text{if } x^i \text{ is the left child of } x^{p(i)} \\ \delta^{i,s} + \delta^{p(i),down}[d+1:2d], & \text{if } x^i \text{ is the right child of } x^{p(i)} \end{cases} \\ \delta^{i,down} &= (W^T \delta^{i,com} + S^i) \circ f'(x^i), \\ \text{where } \circ & \text{denotes the Hadamard product}, & f'(x) = 1 - x^2. \end{cases}$$

The gradients of the error with respect to W, W_s , $V^{[\ell]}$ are

$$\frac{\partial E}{\partial W} = \sum_{i} \delta^{i,com} \begin{bmatrix} x_L^i \\ x_R^i \end{bmatrix}^T, \quad \frac{\partial E}{\partial W_s} = \sum_{i} (y^i - t^i) x^{i^T}, \quad \frac{\partial E}{\partial V^{[\ell]}} = \sum_{i} \delta^{i,com}_{\ell} \begin{bmatrix} x_L^i \\ x_R^i \end{bmatrix} \begin{bmatrix} x_L^i \\ x_R^i \end{bmatrix}^T. \tag{4}$$

3 Techniques for Batching Computation

The existing code in Stanford CoreNLP trains RNTN with mini-batch adaptive gradient descent (AdaGrad) [3], where the gradients are computed one training example at a time, and the forward/backward propagation are computed one node at a time. For these computations, we observe that in the formulae (1, 2, 3), the parameters (W, W_s, V) are shared across training examples in a mini-batch and across nodes.

For foward propagation, we say a node x^i is ready to compute if both x^i_L and x^i_R have been computed. For backward propagation, we say δ^i is ready to compute if the errors of its parent node, i.e., $\delta^{p(i),s}, \delta^{p(i),com}$, and $\delta^{p(i),down}$, have been computed.

We may group ready nodes across trees and compute them all together at once by using the following formulae.

3.1 Group matrix-vector multiplications

After a rearrangement of indices, let x^i , $i=1,\cdots k$, be the nodes that are ready to compute and we have

$$\mathbf{u}_i = \begin{bmatrix} x_L^i \\ x_R^i \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} 1 & 1 & 1 \\ \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_k \\ 1 & 1 & 1 \end{bmatrix}.$$

To compute the matrix-vector multiplication part of (1) over all ready nodes, we have

$$\begin{bmatrix} \mathbf{W}\mathbf{u}_1 & \mathbf{W}\mathbf{u}_2 & \cdots & \mathbf{W}\mathbf{u}_k \end{bmatrix} = \mathbf{W} \begin{bmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_k \end{bmatrix}.$$

3.2 Group Bilinear Operations

To compute the tensor part of (1) over all ready nodes, we have

$$\begin{bmatrix} \mathbf{u}_1^T \mathbf{V} \mathbf{u}_1 & \mathbf{u}_2^T \mathbf{V} \mathbf{u}_2 & \cdots & \mathbf{u}_k^T \mathbf{V} \mathbf{u}_k \end{bmatrix} = \text{Flatten}_2(\mathbf{V}) (\mathbf{U} \odot \mathbf{U}),$$

where \odot is the Khatri-Rao product (defined in the appendix), and the matrix $\text{Flatten}_2(\mathbf{V}) \in \mathbb{R}^{d \times 4d^2}$ is obtained by taking lateral slices of the tensor \mathbf{V} and ordering these slices from left to right. See figure 2(b).

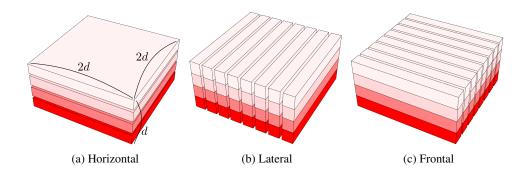


Figure 2: Slice of a 3rd-order tensor

3.3 Group Errors $\delta^{i,com}$ and $\delta^{i,down}$ respectively

To compute the $\delta^{i,com}$ over all ready nodes using (3), let

$$\Delta^{com} = \begin{bmatrix} | & | & | \\ \delta^{1,com} & \delta^{2,com} & \cdots & \delta^{k,com} \\ | & | & | \end{bmatrix} \in \mathbb{R}^{d \times k},$$

then we have

$$\begin{bmatrix} \begin{vmatrix} & & & \\ S^1 & \cdots & S^k \\ & & & \end{vmatrix} = \begin{bmatrix} A^{[1]} & \cdots & A^{[d]} \end{bmatrix} \begin{bmatrix} \delta_1^{1,com}u_1 & \cdots & \delta_1^{k,com}u_k \\ \delta_2^{1,com}u_1 & \cdots & \delta_2^{k,com}u_k \\ \vdots & \ddots & \vdots \\ \delta_d^{1,com}u_1 & \cdots & \delta_d^{k,com}u_k \end{bmatrix} = \operatorname{Flatten}_1(A)(\Delta^{com} \odot U),$$

where $A^{[\ell]} = V^{[\ell]} + (V^{[\ell]})^T$, and the matrix $\text{Flatten}_1(A) \in \mathbb{R}^{2d \times 2d^2}$ is obtained by taking horizontal slices of the tensor A and ordering these slices from left to right. See figure 2(a).

As for $\delta^{i,down}$, we have

$$\begin{bmatrix} \begin{vmatrix} & & & \\ \delta^{1,down} & \delta^{2,down} & \cdots & \delta^{k,down} \\ & & & & \end{vmatrix} = \begin{pmatrix} \mathbf{W}^T \Delta^{com} + \begin{bmatrix} & & & & \\ S^1 & S^2 & \cdots & S^k \\ & & & & & \end{vmatrix} \end{pmatrix}) \circ f' \begin{pmatrix} \begin{bmatrix} & & & & \\ x^1 & x^2 & \cdots & x^k \\ & & & & & \end{vmatrix} \end{pmatrix}).$$

3.4 Group Gradients

We rewrite equations (4) as follows:

$$\frac{\partial E}{\partial W} = \sum_{i} \delta^{i,com} \begin{bmatrix} x_{L}^{i} \\ x_{R}^{i} \end{bmatrix}^{T} = \sum_{i} \delta^{i,com} \mathbf{u}_{i}^{T} = \begin{bmatrix} | & | & | & | \\ \delta^{1,com} & \cdots & \delta^{k,com} & | & | \\ | & & & | & | & \end{bmatrix} \begin{bmatrix} \mathbf{u}_{1}^{T} \\ \mathbf{u}_{2}^{T} \\ \vdots \\ \mathbf{u}_{k}^{T} \end{bmatrix} = \Delta^{com} \mathbf{U}^{T}.$$

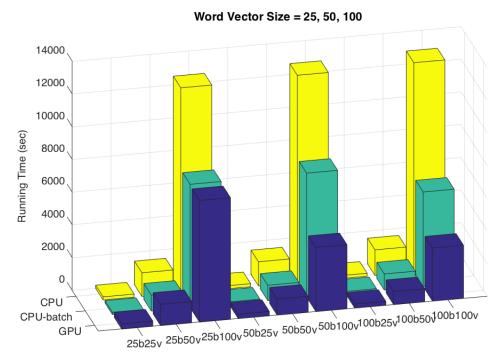
$$\frac{\partial E}{\partial W_{s}} = \sum_{i} (y^{i} - t^{i}) x^{i^{T}} = (Y - T) X^{T}.$$

As for the gradients of the tensor V, we have

$$\begin{bmatrix} \frac{\partial E}{\partial \mathbf{V}^{[1]}} \\ \vdots \\ \frac{\partial E}{\partial \mathbf{V}^{[d]}} \end{bmatrix} = \begin{bmatrix} \sum_{i} \delta_{1}^{i,com} u_{i} u_{i}^{T} \\ \vdots \\ \sum_{i} \delta_{d}^{i,com} u_{i} u_{i}^{T} \end{bmatrix} = \begin{bmatrix} \delta^{1,com} \otimes \mathbf{u}_{1} & \cdots & \delta^{k,com} \otimes \mathbf{u}_{k} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{1}^{T} \\ \mathbf{u}_{2}^{T} \\ \vdots \\ \mathbf{u}_{k}^{T} \end{bmatrix} = (\Delta^{com} \odot U) U^{T}.$$

4 Experiments and Results

We have implemented the above batching techniques as modifications to the "sentiment" module in the Stanford coreNLP project. We use INDArray [5]s from the Nd4j [6] project to represent matrices and tensors. This way it is easy to run the batched operations on a GPU. We ran similar workloads on the three configurations - unmodified coreNLP baseline, batched implementation on CPU, batched implementation on GPU and we share the results below.



Batch Size x Word Vector Size

The specifics of our experiments are as follows:

• We have used git commit id 55bbcb of coreNLP.git as our baseline.

• Nd4j version 0.4-rc3.6

• CUDA/jcublas version 6.5

• CPU: Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz (8 cores)

• RAM: 128GB DDR3

• GPU: GeForce GTX 680

• Dataset: Stanford Sentiment Treebank

• Workload: 1 epoch of training on the Dataset varying batch size (25, 50, 100) and word-vector dimensions (25, 50, 100) in each of the three modes (CPU, CPU-batch, GPU).

• Code: All our code is at http://github.com/avati/coreNLP/commits/rntn-gpu

5 Conclusion and Future Work

Based on the results shown above, we conclude the following:

- Batching computation is always better.
- GPUs offer significant speed-up (up to 4x in our tests) when word-vector dimensions and batch sizes are large enough.
- At lower batch and word-vector dimensions, the overheads of managing data on GPU surpass the benefits of faster computation.

Testing with larger word-vector sizes was interrupted due to a known bug in Nd4j. We are eager to resume testing once the bug is fixed as we expect greater speed-up.

6 Appendix

In this section, we review the definitions of three matrix products [4].

Definitions: Let $\mathbf{A} = (a_{ij}), \mathbf{B} = (b_{ij})$ be m-by-n matrices, $\mathbf{C} = (c_{ij})$ be a p-by-q matrix, and $\mathbf{D} = (d_{ij})$ be a r-by-n matrix. Then

1. Hadamard product:

$$\mathbf{A} \circ \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \cdots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \cdots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \cdots & a_{mn}b_{mn} \end{bmatrix} \in \mathbb{R}^{m \times n}.$$

2. Kronecker product:

$$\mathbf{A} \otimes \mathbf{C} = \begin{bmatrix} a_{11}\mathbf{C} & a_{12}\mathbf{C} & \cdots & a_{1n}\mathbf{C} \\ a_{21}\mathbf{C} & a_{22}\mathbf{C} & \cdots & a_{2n}\mathbf{C} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}\mathbf{C} & a_{m2}\mathbf{C} & \cdots & a_{mn}\mathbf{C} \end{bmatrix} \in \mathbb{R}^{mp \times nq}.$$

3. Khatri-Rao product:

$$\mathbf{A} \odot \mathbf{D} = [\mathbf{a_1} \otimes \mathbf{d_1} \quad \mathbf{a_2} \otimes \mathbf{d_2} \quad \cdots \quad \mathbf{a_n} \otimes \mathbf{d_n}] \in \mathbb{R}^{mr \times n}$$

Acknowledgement

We thank Sam Bowman for introducing the problem and many helpful discussions, Youssef Ahres for mentoring us and offering his expertise when we needed, and finally Prof. Andrew Ng for the education and conducting this wonderful course.

References

- [1] Richard Socher, Alex Perelygin, Jean Y. Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts, Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank, *EMNLP* (2013)
- [2] Christoph Goller and Andreas Kchler, Learning Task-Dependent Distributed Representations by Backpropagation Through Structure, *ICNN* (1996)
- [3] John Duchi, Elad Hazan, and Yoram Singer, Adaptive Subgradient Methods for Online Learning and Stochastic Optimization, *JMLR* (2012)
- [4] Shuangzhe Liu and Gotz Trenkler, Hadamard, Khatri-Rao, Kronecker and Other Matrix Products, *Int. J. Inform. Syst. Sci* (2008)
- [5] http://nd4j.org/apidocs/org/nd4j/linalg/api/ndarray/INDArray.html
- [6] Nd4j http://nd4j.org