Thomas Mayo-Smith
Final Writeup

Summery
My project consists of implementing a parallel recurrent neural network (RNN) on the Latedays cluster machine (Xeon Phi co-processor) then comparing its performance to that of an identically implemented RNN running on a GPU.

Code To Follow Along
Parallel RNN implemented for Latedays Computer Cluster Node:
https://github.com/GeneralMayo/rnn (Written by myself)
GPU compatible RNN and Data Used for this Project:
https://github.com/dennybritz/rnn-tutorial-rnnlm/ (Written by deep learning enthusiest Denny Britz who is not in 15-418)

Background for RNN and Language Modeling Algorithm
RNNs are a class of neural networks designed to learn patterns in sequential data which have dependencies between its values. For instance, if you've ever texted someone on a smartphone you've likely noticed words are auto completed for you. This task could easily be done via an RNN which is basing the prediction off the words you've already typed along with pre-trained parameters which model the flow of the language you're typing in. One important idea to note about this example is the way a RNN can look back at its history to aid with future predictions. This is in contrast with say a normal neural network where inputs are assumed to be independent of one another.

Through the process of tens of thousands of iterations of **1) forward propagation,** where sequences are attempted to be predicted and **2) back propagation,** where errors made by the RNN are attempted to be corrected, the parameters of the RNN slowly converge to a state where they are able to detect patterns in the types of sequences used to train it.

The most basic form of a RNN is as shown below.

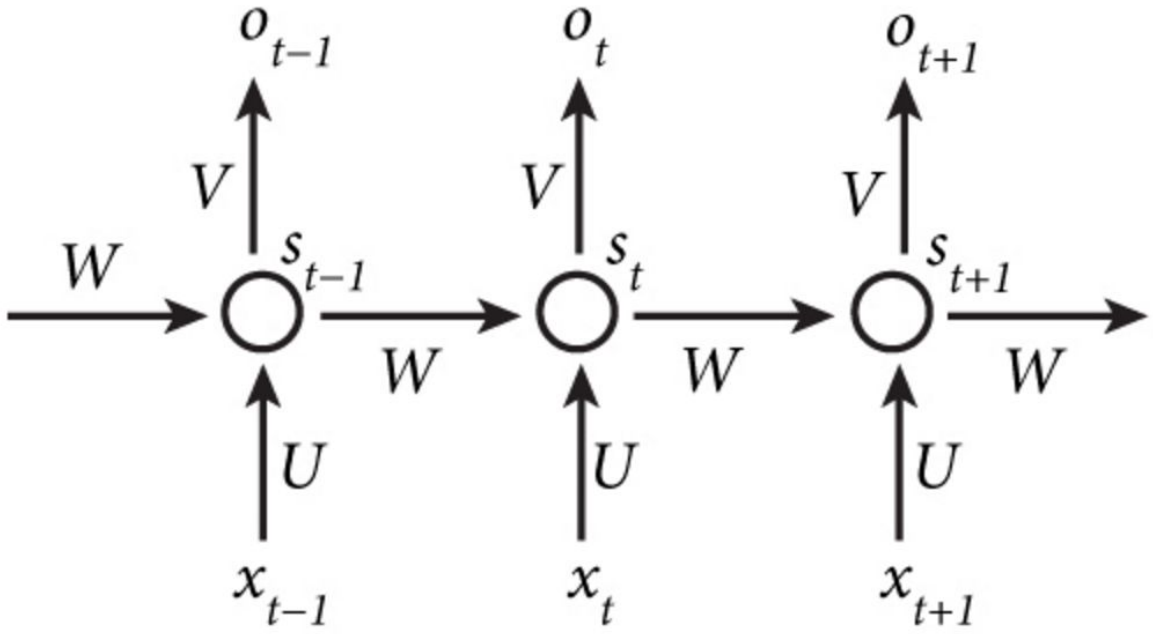


Image Source:
http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-2-implementing-a-language-model-rnn-with-python-numpy-and-theano/

In the above image X = [x_0,…,x_(t-1), x_t, x_(t+1),..., x_(last elem)] can be considered a single training example and is the input of the RNN. To be clear a single training example is X which is a sequence of values NOT x_t or x_(t-1). Thus to train a RNN you need tens of thousands of sequences of type X.

For example, in the RNN I parallelized, X could be equal to something like the following.
X = [If, you, had, fun, you've, won]. In this case, once the RNN is fully trained it may get an input example like [If you, had, fun, you've, ____] and still be able to predict the last word of the input sentence should have been "won" because P("won" | if, you, had, fun, you've) is very high. One important note is that in my specific implementation I only "back propagate through time" 4 time steps so the actual probability my RNN would pick up on is P("won" | if, you, had, fun).

Now the question is how do you represent words in a sentence such that they can be propagated through an RNN? The strategy implemented in my code is based off the the strategy used in the RNN implementation written by Denny Britz. Essentially each word is represented as a “one hot vector” where every element in the vector is a zero except for the element at the index dedicated to the word being represented. This one hot vector is the size of the vocabulary the RNN is trying to learn.

For example, if the sentences trying to be learned by the RNN are…
[Go, home, you’re,a,monk.]
[A, cow, jumped, over, my, home]
…then I would have an index to word mapping such as…
[Go, home, you’re, a , monk, cow, jumped, over, my]
… and the one hot vector of for instance “cow” would be…
[0,0,0,0,0,1,0,0,0]

Meanwhile, the parameter O = [o_0,…,o_(t-1), o_t, o_(t+1),..., o_(last elem)] is the output of the RNN this. The output is a sequence of probabilities generated via a softmax function as below.

$$o_t = \text{softmax}(Vs_t)$$

With sufficient training, an RNN which received the inputs [Go, home, you’re,a] and has an index to word mapping of [Go, home, you’re, a , monk, cow, jumped, over, my] should output the following vector for the last time step.

$$o_t = [.01,.01,.01,.01,.01, .91,.01,.01,.01,]$$

Here a high probability is put on the next word being “monk”. Note how o_t is a vector and O = [o_0,…,o_(t-1), o_t, o_(t+1),..., o_(last elem)] is a matrix.

The “s_t” parameter is commonly referred to as the state of the current input sequence or the “hidden state”. This parameter is calculated based off previous states as well as the current input as shown below.

$$s_t = f(U*x_t + W*s_{(t-1)})$$

“s_t” can be considered the memory of the RNN because of the way it carries information from previous inputs. Also important to note is the size of S = [s_0,…,s_(t-1), s_t, s_(t+1),..., s_(last elem)] essentially dictates the maximum possible history a RNN can have. If for instance an input sequence is longer than the size of S then the first element of that sequence will be independent of the last.

V, W, U are the parameters of the RNN which dictate how values get propagated through the network. In my particular implementation the parameters are the following sized matrixes.

U = 100x8000
V = 8000x100
W = 100x100

These matrices are of type double thus they take up (2*100*8000 + 100*100)*8 = 12,880,000 bytes of memory. Note how len(S) = 100 and Vocabulary_size = 8000. The RNN I implemented is attempting to learn how to use 8000 words in sentences up to 100 words long.

These matrices are updated after each training example is forward propagated through the RNN to correct any error the RNN might have made predicting this most recent training example. The following is a typical parameter update.

U -= learning_rate * dLdU

Note how in this example the gradient dLdU represents the change in “loss” with respect to the parameter U, where loss is an approximation of the errors made by the RNN’s most recent prediction. Gradient calculations for a RNN are complex and won’t be discussed deeply in this report; however, they can be viewed here.

To simplify matters the flow of the entire program is as follows.

For each training example
1) make a prediction based on this training example
2) calculate errors in this prediction with respect to U V and W
3) Update U V and W according to these errors

My Parallelization Methods
First off its important to note that the parallelization methods I experimented with in my project were meant to parallelize the actual training of the RNN. I experimented with the following two methods.

Method 1: Parallelize Across Training Examples Via a Mini Batch Server
Method 2: Parallelize the Most Expensive Processes of Training (Stochastic Gradient Descent)

Method 1 Description
For Method 1 I used OpenMP to parallelize across training examples and Eigen (a C++ template library) for all linear algebra computations. Method 1 was implemented in C++ on a SINGLE Latedays cluster node which most importantly had two, six-core Xeon e5-2620 v3 processors and a 15MB L3 cache.

OpenMP was used to implement a mini batch training algorithm described in the second half of Lecture 26: Parallel Deep Network Training and most notably discussed in the context of RNNs by (Cesar Laurent, Gabriel Pereyra, Philemon Brakel, Ying Zhang, Yoshua Bengio, 2015).

This algorithm is shown in the the following figure.

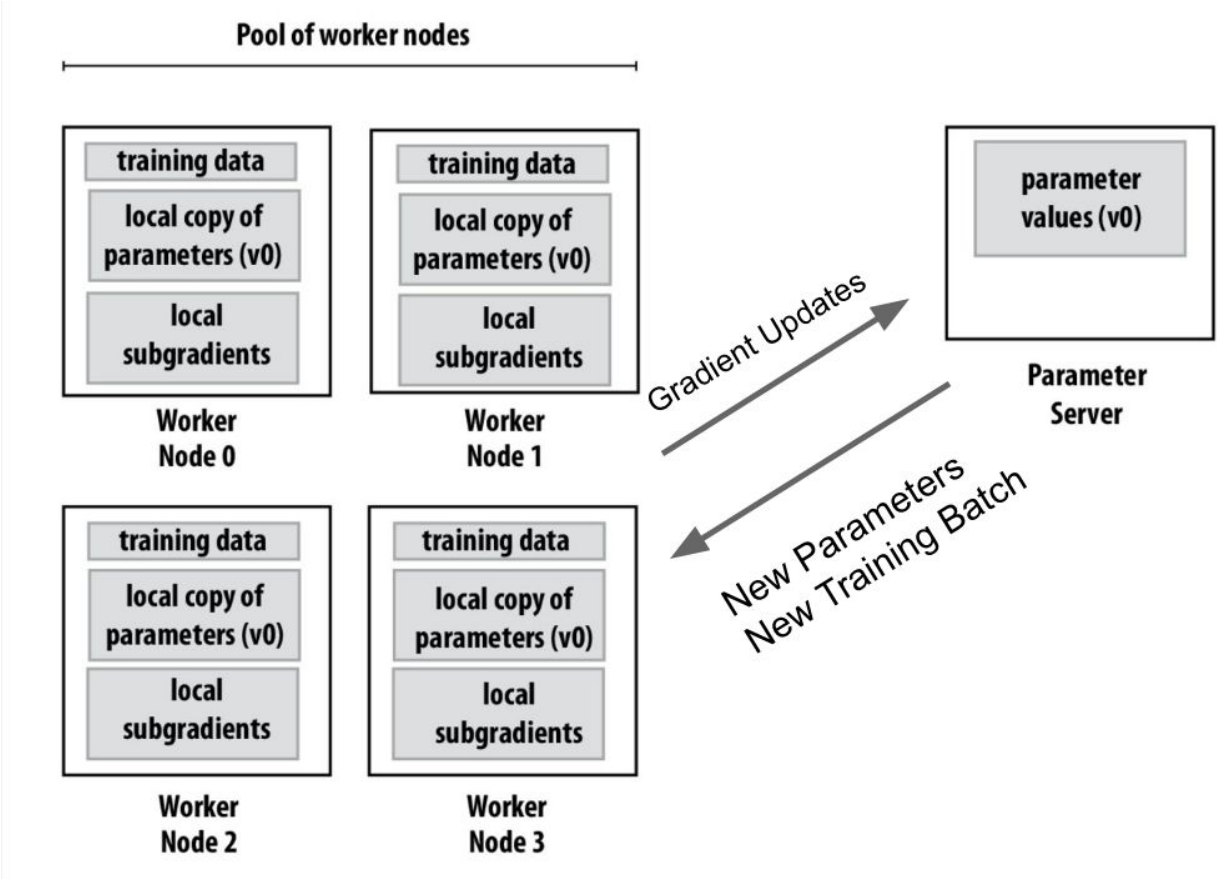


Image Source: Lecture 26: Parallel Deep Network Training Kayvon Fatahalian

In my implementation of the mini batch algorithm each worker node represents a thread which indexes into an array of “parameter structs” and an array of “gradient structs” where it is able to store its local copy of parameters and local subgradients respectively. (Data Structs Below)

```
struct Weights{
        MatrixXd U;
        MatrixXd V;
        MatrixXd W;
};
```

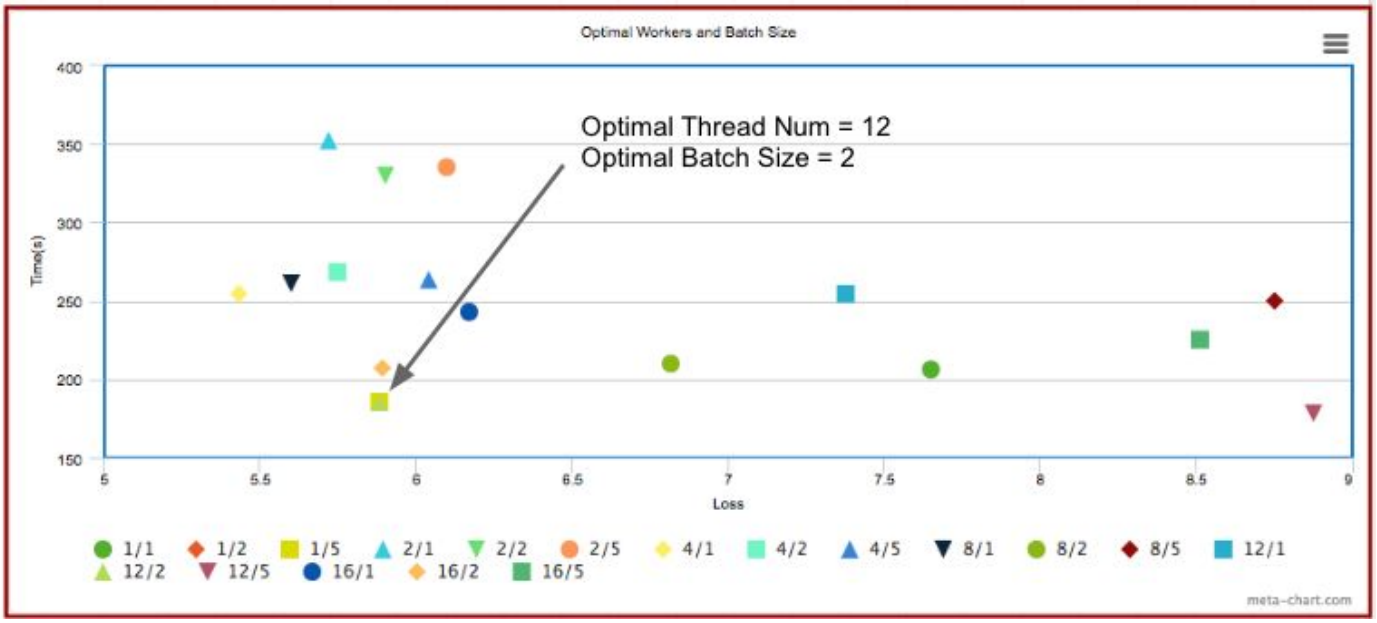
```
struct Gradients{
        MatrixXd dLdU;
        MatrixXd dLdV;
        MatrixXd dLdW;
};
```

```
Weights * worker_weights = new Weights[NUM_THREADS];
Gradients * worker_grads = new Gradients[NUM_THREADS];
```

Each thread is statically assigned interleaved batches of training examples such that they can compute gradients independently and update the parameters in the “parameter server” asynchronously. Updates to the parameter server’s parameters are done inside an OpenMP critical block to prevent parameter corruption. Interestingly from my experimentation I determined the private parameters of the workers didn’t need to be updated within the critical block regardless of the fact these worker parameter updates could be corrupted. For instance, when the worker parameters are updated outside the critical block the loss of the RNN calculated after 5000 training examples consistently does not diverge from the loss of the RNN when the worker parameters are updated inside the critical block.

Another interesting finding along these same lines was the contention around the critical block with and without the worker parameters being updated inside differed dramatically. For instance, when the worker parameters were being updated inside the critical block the fraction of the amount of time threads spent inside the critical block with respect to the amount of time they spent in the back propagation function was 1/4. However; when the worker parameters were being updated outside the critical block this fraction changed to 1/13, which was nearly equivalent to this same fraction when only one thread was being used. The resulting speedup was around 1.5x.

In contrast to this simple optimization likely the most time consuming optimization I made was choosing the optimal batch size and worker number for the mini batch algorithm. After cycling through thread nums {1,2,4,8,12,16} and batch size nums {1,2,5,10} I generated the following data. For each combination of thread num and batch size I tracked how much time it took the RNN to get through 5000 training examples and the loss of the RNN after the training was complete. Below the the actual point plot is the legend which states the THREAD_NUM/BATCH_SIZE.



I hypothesise 12 was the most optimal thread/worker num because each thread utilizes one of the 12 cores on the Xeon Phi co-processor. Although the Xeon Phi supports hyper threading, its likely given the combined memory usage of the parameters of all the worker nodes which was at least greater than 154MB and the L3 cache size of the Xeon Phi which is 15MB, L1/2 and 3 cache conflicts made the utilization of over 12 threads sub-optimal. The results were a 2.2x speedup at the cost of 4% increase in loss.

Method 2: Parallelize the Most Expensive Processes of Training (Stochastic Gradient Descent)
The most expensive process of training the RNN is by far the SGD step due to the fact that gradients for an RNN depend on many parameters used to calculate states and outputs in previous time steps. RNN's are good at finding patterns in sequential data because they can draw upon their history of inputs; however, it is this entire history which can cause error in the current prediction and thus must be taken into account when considering the loss of the current prediction. In short, forward propagation takes on the order of thousands of a second, parameter updates take on the order of several milliseconds and SGD takes on the order of

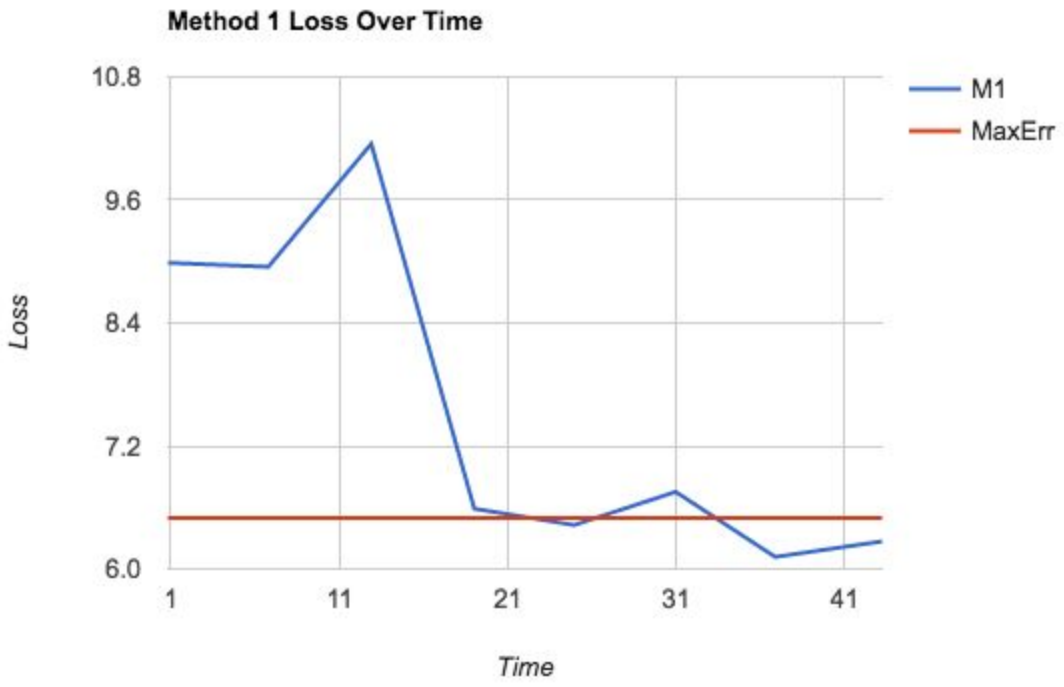
tens of milliseconds. Thus focusing as much effort as possible to parallelize this step is well worth it.
To experiment with the parallelization of this step I used an RNN implementation written in python which utilized the Teano library to perform SGD.

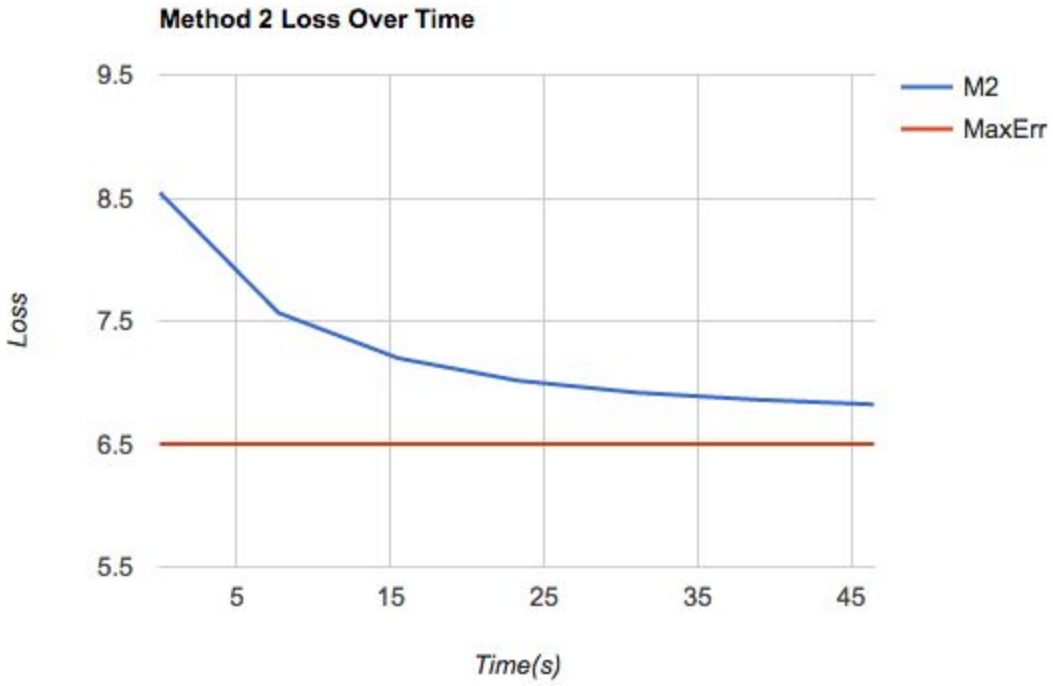
The specific lines of code where SGD is performed are below.

```
self.sgd_step = theano.function([x,y,learning_rate], [],
                    updates=[(self.U, self.U - learning_rate * dU),
                          (self.V, self.V - learning_rate * dV),
                          (self.W, self.W - learning_rate * dW)])
```

I ran this theano code on an Amazon EC2 Instance with a High-performance NVIDIA GPU, with 1,536 CUDA cores.

To compare the performance of Method 1 and Method 2 I ran both implementations on the exact same 5000 training examples, with the same initial learning rate, and the same backpropagation through time steps to see which one would dip below a loss of 6.5 first and stay below 6.5 for at least 2 consecutive epochs. Below are the results.





The RNN with a mini batch server ended up achieving a loss lower than 6.5 for two consecutive epochs faster than the RNN with an optimized gradient descent did.

References

https://arxiv.org/pdf/1510.01378.pdf - Batch Normalized Recurrent Neural Networks