# Autogenerating Text via Parallel Implementations of Recurrent Neural Networks

Taylor Clarke (tclarke)
Anirban Ghosh (anirbang)

## I. Summary

We are going to implement recurrent neural networks on multiple pieces of hardware, such as a CPU, a Xeon Phi, and a GPU, and compare performance between the implementations, as well as compare performance with different parameters of the neural network. Our basic test will be the time necessary to train and create a valid XML page.

## II. Background

Neural networks are a machine learning method to predict an output variable based on a set of input variables. Usually many points are used to "train" the neural network to attempt to compute the correct output, and then the network is tested on other data with known outputs to estimate its accuracy. If the neural network meets the specifications for the precise needs of the developer, it can then be used to predict outputs for data that is unknown. A neural network is essentially a directed graph, where a node computes a function of its input and directs that output either to another node or to be the final output of the network. The functions represented by the nodes can be univariate or multivariate and produce anything from a basic linear function to a binary classifier of its input. With the appropriate choice of functions, a neural network can be trained to produce a wide range of outputs.

The functions represented by a node in the neural network often require parameters. These parameters are determined by training the neural network. This consists of computing the predicted output for known data, comparing that output with the actual output, and utilizing some method of tweaking the parameters, such as error backpropagation via gradient descent, to minimize the error between the predicted and the actual output. With a well-designed network and proper input data, this error will presumably decrease until the accuracy of the neural network is as high as possible.

Our specific interest is a recurrent neural network, which is merely a neural network where some of the output of the nodes are fed back to nodes previously visited, ie there is a cycle in the network. Recurrent neural networks are often used when we suspect that the output is correlated to other outputs in the dataset, so that utilizing a function of previous input and output will give information about the next output. One common way to visualize this is a time series, such as stock data. We expect the price of a stock at a certain time of day to depend not only on the time of day but also on the prices at previous timesteps. This also means that recurrent neural networks are an obvious tool for natural language processing, where previous characters in a word have some effect on the next character, previous words in a sentence have some effect on the next word, and so on. This is the foundation for our task. We will train the network with a large amount of XML data, and the network will adjust its

parameters to predict the next character as a function of the previous characters. The goal of this process is that upon seeding the network or relying upon the network to seed itself, we allow the network to use its own output as the next input, so it could theoretically produce a large amount of meaningful text, code, or data on its own.

Parallelization of neural networks has been a hot topic in academia and industry for a number of years. As with any graph problem, parallelization is not obvious or easy, but can lead to great speedups when implemented cleverly. The major issue with parallelizing neural networks is that in order to train the network effectively, errors must be backpropagated through the network to compute the parameter alterations. This means that information must be passed both forward and backward through the net, leading to a large amount of interdependence and communication in the process. However, if independent nodes can be computed in parallel and data communication costs can be minimized, a parallel algorithm has potential to be much faster than a sequential algorithm.

## III. The Challenge

There are existing tools to implement neural networks, even recurrent neural networks, on both CPU's and GPU's (see the Resources section for specifics). However, since these tools are highly generalized, they may not produce adequate performance for specific tasks. For instance, a CPU may have faster processing speed and more RAM, but less availability to run threads asynchronously, while a GPU is able to be much more highly parallel but probably has less RAM available. Therefore, as is often the case with parallel computation, before implementing a specific task it is somewhat unclear how a specific machine will perform on the task. Our task specifically is to determine the variations in performance between multiple hardware implementations on training and using a neural network to produce valid XML. It is unclear at this point how much training data is necessary or what design of the neural network will allow the network to most quickly converge, and these decisions could be different for different hardware implementations. It is our goal to determine the optimal network design and training schema for each hardware implementation and then compare performance between them. Specifically this task will most likely involve much intercommuncation between different nodes of the neural network, so there is a high communication-to-computation ratio, which we may have to tackle with different methods for different pieces of hardware. Additionally, the amount of text we might have to train on may be too much or too little to efficiently implement on a CPU or GPU. These specific issues will determine the hardware most suited to solve our problem.

## IV. Resources

a. Hardware resources:
    i. Multi-core Intel CPU based workstation/servers for basic multithreaded software implementation.
    ii. A machine with an Intel Xeon Phi processor with OpenMP for the Xeon Phi implementation.
    iii. A machine with an NVIDIA GeForce GTX1080 GPU with CUDA for benchmarking the GPU based implementation.

b. Software and code bases:
    i. The existing code base and data sets that we will be borrowing from can be found here: (http://karpathy.github.io/2015/05/21/rnn-effectiveness/). This is a basic implementation of a recurrent neural network using characters as input, which is what we require for our task. We may alter this codebase in order to optimize performance for various pieces of hardware.
    ii. We plan to use the Torch framework for the implementation. Torch is a scientific computing framework built in Lua, C, and CUDA that is commonly used for machine learning tasks.

**V. Goals/Deliverables**

Our goal is to benchmark the performance of multi-core CPUs, multi-core CPU-based compute accelerators and GPUs for a common machine learning algorithm (recurrent neural networks). We plan to utilize some existing models/datasets which have known results and first benchmark them for validity and performance on regular CPUs. We will then migrate this code to an Intel Xeon Phi accelerator and explore different opportunities for parallelism as well as benchmark the Phi's performance. Finally, we will port the code to an NVIDIA GPU using CUDA and compare the obtained speedups for the Xeon Phi and the GPU. At each point we will attempt to optimize the algorithm as much as possible by potentially altering the design of the neural network or its parameters in order to converge faster or produce more accurate results for that specific hardware implementation. As a moonshot goal, we hope to be able to deploy this in a multi-CPU or multi GPU cluster and study the challenges with scaling machine learning systems across clusters, and hopefully implement further optimizations to improve speedup on a cluster. For our final results, we will be able to show the runtime differences between different hardware implementations, and potentially show how our trained network can be used on a common desktop to autogenerate XML data.

**VI. Platform**

We have chosen to benchmark and compare multiple platforms that implement parallelism in different ways in order to study how suitable different architectures are for recurrent neural network applications. The first platform where we will develop a basic software implementation and baseline is a standard CPU. This will provide the easiest development platform, and the naive approach, while slow, will help establish the expected results for when we parallelize the application. Since most modern CPUs have multiple cores and multithreading, it will be easy to study different axes of parallelism on CPU and develop ideas for parallelizing the future stages.

The next platform we will port our application on is the Intel Xeon Phi coprocessor. The Xeon Phi was chosen because it has a large number of simpler x86 CPU cores that are ISA-compatible with regular CPUs, making the code porting process a natural transition from a CPU environment using OpenMP. This leaves us free to further explore how our algorithm scales rather than focussing on code syntax at this stage. The other advantage of the Xeon Phi is the fact that unlike the SIMD nature of GPU architectures, the multi-x86 core architecture

allows for far more complex, even divergent code to be written and run in parallel. We want to see if this can be leveraged in our application.

The third and final platform we want to test in this application is a GPU running CUDA. GPUs provide massive SIMD parallelism with potentially thousands of simpler execution units capable of running in parallel. GPUs have become the platform of choice for modern day machine learning algorithms, and we want to benchmark the overall speedup they provide over CPUs and CPU compute accelerators like the Phi.

**VII. Schedule**

| | |
|---|---|
| Week of 10/31 - 11/7 | Learn the basic implementation, study literature and implement a basic (naive) neural network on local machines. Validate results to set expected correct outputs. |
| Week of 11/7 - 11/14 | Explore parallelism by implementing multi threaded, parallel versions of the code on local machines. |
| Week of 11/14 - 11/21 (Checkpoint) | Port code to the Xeon Phi and continue to study and optimize for parallel execution. By the checkpoint we hope to have a baseline result with a CPU and results with the Xeon Phi to directly compare the effects of higher parallelism on our specific algorithm. |
| Week of 11/21 - 11/28 | Port code to the GPU and benchmark results v/s CPU and Xeon Phi. |
| Week of 11/28 - 12/5 | Explore optimizations to improve prediction quality as well as computation time. If training of the network has been successful, potentially port the network to a basic desktop to show how the network can autogenerate XML data. |
| Week of 12/5 - 12/12 | Final benchmarks, validation, reports. |