Modelling Airline Delays Using High Performance Artificial Neural Networks

Alfredo Luque

***Abstract*—Bureau of Transportation Statistics on-time arrival (OTA) datasets contain a wealth of information pertaining to causes for airline delays for all commercial U.S. flights. On its own, this dataset does not provide sufficient information to develop predictive models for delays. Here, I join the OTA dataset with hourly weather data published by the National Oceanographic and Atmospheric Administration (NOAA) and airport operations data published by the Federal Aviation Administration (FAA) in order to develop an accurate predictive model for commercial flight delays.**

Given the complex nonlinear relationship between the variables I utilize a feed-forward multilayer perceptron artificial neural network (FF MLP ANN) to approximate the relationships. Even using high-order optimizations to traditional backpropagation techniques, convergence requires significant computational time; hence, I develop a GPU implementation of backpropagation with momentum that sacrifices some algorithmic efficiency for massive parallelism. Testing on a select number of major airports in the U.S. reveals a high level of predictive accuracy.

I. Introduction

COMMERCIAL flight delays in the United States have been recorded by the *Research and Innovative Technology Administration* (RITA) since 2005. Although contribution of the data is not mandated by law, in recent years the database has included the vast majority of commercial U.S. flights. This data set includes, among other fields, flight distances, causes of delays, delay breakdowns by cause, origin and destination airports, and synthetic variables derived from the data.

Traditional flight-delay reporting systems use either (1) FAA reported scheduled departure times or (2) Carrier-based information systems. Changes scheduled departure times filed with the airport usually only occur a few minutes before the original departure slot (or sometimes after). Similarly, internal scheduling systems often tend to report delays only a few minutes before a flight is scheduled to leave (or if elevated airport congestion introduces a necessary delay). A prediction system that does not rely on real-time air traffic information can provide guidance this much earlier, at the expense of being able to predict abnormal delays caused by mechanical problems or 'acts of god'. Since most delays are a result of either weather or airport congestion,data sets that provide this information can be used to construct a predictor for flight delays.

In order to deliver accurate predictions, high-resolution data is necessary. In the U.S, NOAA provides surface data in daily, hourly, 5-minute, or 1-minute intervals. While the 1-minute data generated by the Automated Sky Observation System (ASOS) provides the highest resolution, they provide less detailed information about cloud heights that can cause delays by restricting landings to runways dedicated to ILS (Instrument Landing System) approaches. The 1 hour data (Quality Controlled Climatological Data) provides sufficient resolution to predict weather-related delays and is provided in formats that are substantially easier to parse.

Airport congestion is difficult to measure directly; however, *ceteris paribus* total air operations (departures and landings) are a good substitute to use in a model. However, since congestion can also be affected by weather, it is important that the predictor allow for interactions between these two inputs. The FAA publishes daily totals of commercial air operations for most major U.S airports.

II. Model Specification

If $\vec{\mathbf{X}}$ is a vector containing distance, temperature, pressure, windspeed, visibility, commercial air operation totals for the airport, and cloud height then I seek a mapping from $\mathbb{R}^{\text{IN}} \to \mathbb{R}$ that predicts $(\beta \cdot \text{Delay})$ where β is a scaling parameter used to constrain the range of 'Delay' to an interval approximating $[0, 1]$.

A feed-forward neural network allows for this type of mapping. Consider the simplest such arrangement:

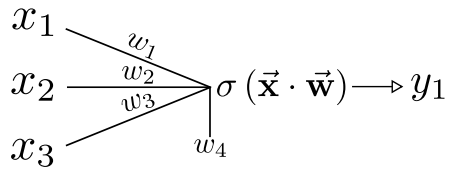

Fig. 1. Single Neuron

Here, the output of the network is $\sigma\left(\sum_i x_i w_i\right)$ where $\sigma(x) = (1 + e^{-x})^{-1}$, a sigmoid activation function chosen due to its ability to compress outputs in the range $[-1, 1]$ and easy to compute derivative. This single neuron[1] can map a linear relationship $\mathbb{R}^3 \to \mathbb{R}$ since it has a bias term that allows for lateral shifts in the data.

Such an arrangement, however, prevents interactions from being considered. Hence, adding an intermediary layer allows the network to approximate any continuous function between $\mathbb{R}^{\text{inputs}} \to \mathbb{R}^{\text{outputs}}$ or more generally between any two n-dimensional vector spaces. In practice, for a single 'hidden'

[1]Note that a neuron is the continuous analog of a perceptron which is a binary classifier.

layer to be effective, the dimension of the output vector space should be significantly less than that of the inputs. Predicting airline delays ostensibly maps to $\mathbb{R}$ which allows for an accurate network to contain only a single hidden layer.

A. Selecting a Topology

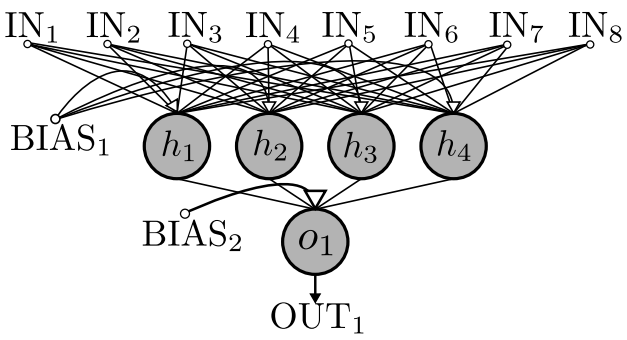

Fig. 2. Multilayer Perceptron Network

Since each additional neuron and layer add complexity, it is important to select the simplest possible network topology that accurate captures the problem. In addition, creating too complex a network will result in significant overfitting. After implementing the network in Figure 2 using the PyBrain module for Python, I attempted the following variations:

1) Adding an additional hidden layer
2) Adding 1-2 more hidden neurons
3) Subtracting 1-2 hidden neurons

Ultimately, 4 hidden neurons converged in a reasonable amount of time. While two layers eventually converged to a very low error, this topology resulted in overfitting when errors were computed for a test set. Generally, two layers are only required in problems with unique discontinuities. Implementing direct connectivity (adding weight vectors from the inputs directly to the output) did not significantly change the final errors. This indicates that there is some degree of linear separability in the data. For my final topology, I tested a $9:4:1$ and $9:5:1$ network.

Using the general topology described above, the overall network function then takes the form:

$$o_k(\mathbf{X},\mathbf{W}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)}\sigma\left(\sum_{i=1}^{D} w_{ji}^{(1)}x_i + w_{j_0}^{(1)}\right) + w_{k_0}^{(2)}\right) \tag{1}$$

Since processing occurs in two stages that as a whole resemble the operation of a simple perceptron, this type of neural network is known as a multilayer perceptron.

B. Training The Network

Computing the optimal $\mathbf{W}$ is more difficult. We wish to minimize an error function (typically sum of squared deviations from expected outputs) by choosing optimal weights.

This will happen when $\nabla E(\vec{W}) = 0$. Given this condition, finding the minimum on the error surface directly is impractical (and often impossible) so an iterative numerical approximation procedure is used instead. I utilize the backpropagation algorithm, which calculates δ for each neuron by first calculating forward activations and then propagating errors backwards throughout the network. Assuming a sigmoidal activation function, for each neuron we calculate:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\cdot\sigma(a_j) = \delta_j\cdot\sigma(a_j)$$

where a_j is an activation. For an output neuron, $\delta_j = y_j - t_j$ where y_j is the j^{th} output from the network and t_j corresponds to the target value for the j^{th} output.

The chain rule allows us to compute:

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_j} \tag{2}$$

$$= \sigma'(a_j)\sum_k w_k\delta_k \tag{3}$$

$$= \sigma(a_j)(1-\sigma(a_j))\sum_k w_k\delta_k \tag{4}$$

The final formulation allows for derivation of errors in a given layer in terms of errors in the layer directly after it (i.e. closer to the output). Moreover, given that $\sigma(x)$ has such a simple to compute derivative, it's possible to evaluate (3) without significant computational overhead.

For a given weight in update period T, the update is computed by:

$$\Delta w^T = \eta\cdot\delta a \qquad \eta\in[0,1]$$

Where η is a 'learning rate' that allows for a gradual descent to a minimum. In order to prevent oscillations around local minima and speed up convergence, we can modify the update formula so that weight changes tend to move in the same direction. To do this, we add a momentum term:

$$\Delta w^T = \eta\cdot\delta a + \xi\cdot\Delta w^{T-1} \qquad \eta,\xi\in[0,1] \tag{5}$$

This procedure is then performed iteratively until errors begin to converge.

III. Implementation

In its simplest form, a neural network is defined by a matrix of weights. To activate the network, we take a vector of inputs and propagate it through the network using Equation 1. In general then, the procedure requires finding $\vec{\mathbf{X}}\cdot\vec{\mathbf{W}}$ for each layer, passing that value through the activation function, and repeating for all remaining layers. After computing the error from the output, this is then reversed and δ's are calculated for each neuron. In the single-threaded case, the implementation is trivial and requires a few nested loops. Weights are updated after each pattern. One epoch is then one successful pass through all the input patterns.

This approach is inefficient since a very large number of simple floating point operations occur sequentially. In essence,

a single thread's ability to perform all the operations becomes the bottleneck. Neural networks inherently feature a high degree of parallelism; however, communication is necessary to ensure that a previous layer's operations are complete before attempting to compute dot products at the subsequent layer. That is, layer $l+1$ is dependent on layer l's activations having been computed.

A. CUDA Overview

Given the need for communication and/or synchronization between threads, traditional map/reduce style frameworks are not appropriate for efficiently training neural networks.Since at present most CPU's feature no more than 8 cores, parallelism is relatively limited. Graphics processing units (GPU) feature thousands of 'cores' that can perform simple operations on a massively parallel scale. In addition, each core features extremely fast on-chip memory that allows threads to share data. The challenge then becomes maximizing throughput.

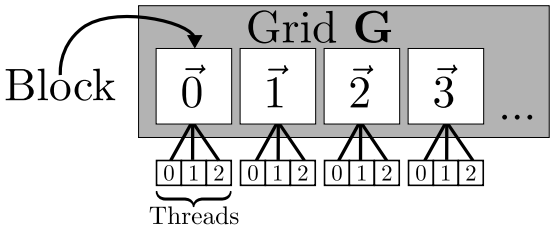

Fig. 3. NVIDIA® CUDA™ Computational Hierarchy

While multiple methods exist of performing operations on GPUs, the CUDA computing platform features an easy to use C++ API and a flexible programming model. In short, operations are divided into grids, blocks, and threads. Grids and blocks can exist in two dimensional space; however, for my purposes a single 1-D grid is sufficient. Blocks cannot communicate with each other; however, their threads can use fast shared memory and can synchronize. Generally, computation in CUDA is achieved by invoking 'kernel calls'. A 'kernel' is simply a program that is executed on the GPU and is called from the CPU (host). In addition to kernels, we can specify device functions that will execute only on the GPU and can be called from any running kernel (but not from the host machine).

B. In-Network Parallelization

Choosing how to structure the network using CUDA is far from trivial. Ideally, operations that require synchronization reside in threads. It makes sense, then to implement a single network in a block. In the next section, I will describe how these blocks can then be further parallelized.

We can create one thread for each of n neurons in a layer and simply move from layer to layer. This process requires synchronizing threads after the computations for each layer are complete. Ostensibly, this introduces some overhead if the processing time for each thread diverges. A few key modifications ensure this doesn't happen.

1) Scaling all inputs to be in [0,1].
2) Using single precision floating point numbers.
3) Using as few `synch_threads` requests as possible.

I choose n to coincide with the number of hidden neurons (the bias term is simply added on to each partial sum from the previous layer). This ensures that all threads are busy for the majority of the time. When computing the output weights, HN-ON threads will be idle; however, given that I use only one output neuron and a small number of hidden neurons, this period of time is minimal.

Consider, then, the following layout:

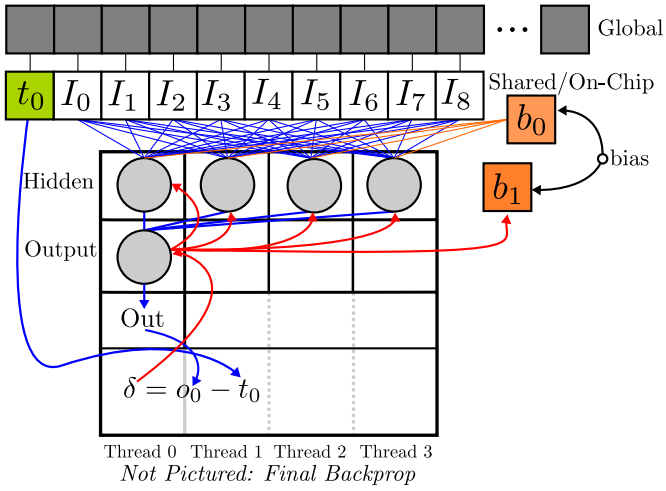

Fig. 4. Neural Network Parallelization Using Threads

This layout ensures that during the most computationally expensive process (going between input and hidden layers or vice-versa) we have full parallelization. However, on a GPU with hundreds or thousands of CUDA cores this still results in low utilization. The solution, then, is to sacrifice the efficiency of on-line stochastic gradient descent for a method that allows mass parallelization.

C. Batch Training

Instead of updating weights for every pattern, we update them every epoch (one full sweep of all the patterns in the training set).

For each weight w_{ij} we compute Δw_{ij}^t for all patterns t. Then the final weight change is simply:

$$\Delta w_{ij} = \sum_{t=0}^{T} \Delta w_{ij}^t \tag{6}$$

This allows us to compute the weight updates for each pattern independently and simply add them up at the end. Assuming each pattern takes the same time to propagate and backpropagate, this means that hundreds or thousands of patterns can be processed in the time that it would take to process one without batch training. Assuming that the training set has cardinality $2^n, n \in \mathbb{N}$ then we can achieve full GPU utilization for at least a portion of each epoch.

The final implementation thus invokes a kernel on one block for each pattern in the training set, each with one thread for each neuron in the hidden layer. Note that if we launch too many kernels, the GPU will simply compute as many blocks as possible first and then process the remaining blocks. With a large enough training set (in the tens of thousands) this idle time is negligible. Also note that convergence occurs more slowly with a batch method as opposed to on-line training.

D. Performance

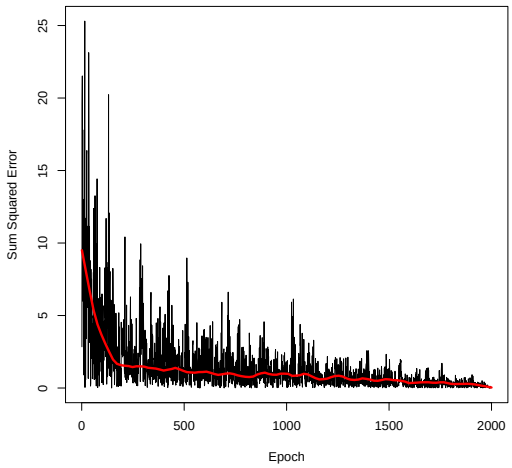

Fig. 5. Convergence of online backprop algorithm

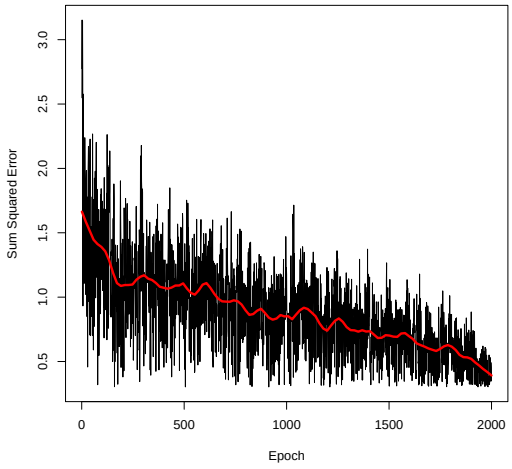

Fig. 6. Convergence of batch backprop algorithm

In Figures 5 and 6 we see that batch training produces a more gradual slope and is less sensitive to converging at a local minimum (in Figure 5 convergence occurs at an error of approximately .75). However, the biggest difference is in performance. While it took the online algorithm in Figure 5 2 hours to run through 2000 epochs, the CUDA enabled algorithm in Figure 6 took only 4 minutes 49 seconds using dual NVIDIA Tesla M2050 GPUs[2]. This is a speed up of nearly 25x over a single-threaded implementation.

In practice, the speedup is not as large since it took the batch CUDA implementation more than 2000 epochs to converge. The actual speedup is 10-15 % less based on my estimations. It should be possible to increase performance significantly by utilizing texture memory to store weights and optimizing the algorithm for sequential reads. Differences between GPUs can also have massive effects on performance. While Amazon EC2 uses NVIDIA Tesla M2050 GPUs (1.03 TFLOPS), testing the algorithm on workstation with a GTX 680 GPU[3] (3.09 TFLOPS) yielded speeds roughly 10-15x faster. Perhaps this is due to overhead from the virtualization used in the instance.

IV. Results

Given the difficulty in processing data, I was only able to train networks on data from four major airports: New York Laguardia (LGA), Chicago Midway (MDW), Chicago O'Hare (ORD), and Los Angeles (LAX). These results are surprising. For all of the airports tested, it is possible to predict delays within 10 minutes using just weather, flight info, and airport congestion data. LAX is peculiar in that even with a more complex network, it was difficult to attain convergence. Further examination of LAX's flight delays shows extremely high variance and some delays that span several hours. I modified my implementation to use a $9:5:4:1$ topology and attempted to fit a model for LAX again. This time, the average error went down to 5 min and the variance to 10. However, testing the final model on data from 2011 revealed issues with overfitting. Perhaps additional data is necessary in order to better estimate delays at LAX.

TABLE I. Accuracy of Delay Predictions

Airport	Patterns	Avg Error (m)	Var(Error)	Time (m)
Laguardia (LGA)	16,384	2.519	6.829	10.6
Los Angeles Int. (LAX)	16,384	10.855	28.290	18.2
Chicago Midway (MDW)	32,768	1.157	7.114	20.1
Chicago O'Hare (ORD)	32,768	3.545	4.262	

A. Extensions

It should be possible to use carrier data to better estimate delays due to mechanical problems. Since these delays are inherently stochastic, any differences between carriers would be due to variations in fleet composition and maintenance schedules. For large airports like LAX, congestion could be

[2] Amazon EC2 GPU Cluster Compute Instance running Ubuntu Server 12.04 HVM LTS with CUDA 5.5 Preview Release

[3] This GPU was used for final training

best estimated by separating total air operations into international and domestic flights. International flights are usually given landing priority so a large number of scheduled arriving international flights could be an indicator of airport-wide delays.

With these data extensions, there may be benefits from augmenting the network topology to include two hidden layers as I experimented with using LAX data. Unfortunately, adding these extensions increases training time significantly. That said, it should be possible to reduce training time considerably by using information from higher order derivatives in the backpropagation procedure. These optimizations attempt to compute the Hessian matrix which describes the second order nature of the error function's surface[4]. Assuming all of these improvements, it would not be inconceivable to train networks for all major airports in the U.S. Given the scope of training data, networks would only need to be retrained at most once per year. Moreover, each network can be stored as a small matrix of weights and can be evaluated efficiently on any hardware.

APPENDIX A
CODE ORGANIZATION

1) `nn.cu`
   Contains a kernel for network activation and the backpropagation procedure. Also contains device-only mathematical functions used for computations. This is really the heart of the program and contains everything necessary for the network to train itself
2) `array_reduction.cu`
   Contains a highly optimized kernel for adding up the elements of a 1D array using sequential reductions. This is a single precision float adaptation of the algorithm mentioned here.
3) `main.cu`
   Contains the host control code. Launches device kernels. Changes can be made here to control what do with weights, etc.
4) `csv_v2.h`
   Fast CSV parser.Source
5) `vecadd.cu`
   Cuda enabled vector addition for floats (adapted from CUDA by example)

Due to the current nature of CUDA, quite a few changes are required at compile time in order to generate a network. Specifically, the topology and how to handle the outputs. In addition, it's important to specify whether or not to output sum squared errors. Compilation involves individually compiling all files then linking the CUDA fast math libraries, linking the objects, and specifying which compute capability to use for execution (if compilation included multiple PTX variants) .

[4]An interesting read touting some of the benefits of using second-order methods in hidden layers is Vatanen, Raiko, *et al.* 2013