OPTIMIZING THE MULTICLASS PERCEPTRON THROUGH PARAMETER TUNING AND GPU UTILIZATION

Shayne McIntosh, Gary Zoppetti, Stephanie Elzer Schwartz
Millersville University
sdmcinto (at symbol) millersville (dot) edu

ABSTRACT
Artificial neural networks are biologically influenced models which serve as a common tool for data analysis. They produce state-of-the-art results in machine learning fields such as computer vision and speech recognition. Artificial Neural Networks are complex probabilistic models. Training them is computationally expensive. Consequently, optimization and trade-off management are paramount. Optimization methods can generally be arranged into two groups, algorithmic advantage and computing power or hardware advantage. This paper presents data associated with simple optimization methods from both groups for a multinomial logistic regression network. GPU utilization provides hardware or computing power advantage and parameter tuning methods provide algorithmic advantage. Learning is conducted on the MNIST data set. Its assessment is meant to provide intuition for novice network optimization and training.

KEY WORDS
Cost, hyperplane, epochs, learning rate, batch size.

1. Introduction

The origins of artificial neural networks (ANN) date back to 1943 when McCulloch and Pitts developed a computational model based on neurons and their connections in the brain [1]. Their invention shortly followed the advent of digital computers. At the time, even the world's best super computer lacked the power and memory necessary to train artificial networks. Consequently, ANNs remained a theoretical concept, unable to be implemented or tested until computational power evolved sufficiently.

Today, even personal machines can achieve reasonable performance training smaller scale networks. High performance machines are capable of massive scale data computation and are maintained by companies like Google, Amazon and Facebook. In fact, Facebook has promised to open sourced their GPU run server design for anyone to use. Their strategy is to drive algorithmic innovation by providing top performance machines to as many data scientists as possible [16].

Despite growing availability of high performance machines, the problem of optimization remains. The scale or size of networks is typically quantifiable through parameter or input volume since it provides a robust measure. Consider today's largest, most complex convolutional networks and the systems in which they're trained. In 2013, Coates et al. developed a system capable of training a network with as many as 11 billion parameters [2]. Their record was shattered in 2015 when a group of researchers at Digital Reasoning developed a system which utilized parallel training methods in order to train a 160 billion parameter network [3]. Networks of this size are most effective for AI problems but at an overwhelming cost. The greater the size, the greater the complexity, the greater the time spent training them. Many days can be spent waiting for a computer to churn out a trained model. Our problem grows as we multiply this time across the number of iterations necessary for minimizing error rates. As a result, a sequence of trial and error on one type of network can be drawn out over many weeks or months. We see that despite our relatively new ability to utilize ANNs within AI, original limitations have persisted and will continue into the unforeseeable future. For this reason, much focus within AI research is spent on optimization of neural networks and the systems in which they are trained.

This paper focuses on baseline methods for machine learning optimization. Ultimately it quantifies the performance gains attained by tuning two parameters and leveraging a GPU.

For clarity in experimentation we optimize on one of the simplest ANNs. Learning is conducted on the famous MNIST handwritten digit data set. Theano, a Python compiler and library, is used for optimizing mathematical expressions on CPUs and GPUs [6][7].

To start, the paper will introduce the MNIST classification problem. In following, it provides context by describing each component of the multiclass perceptron and the classification problems it is meant for. Finally, it presents our generated data showing the affect of parameter tuning and GPU utilization on performance for our model.

2. MNIST Classification

The famous MNIST data set used for machine learning was developed by Yann Lecun and previously Corinna Cortes and Chris Burges. It is composed of 28x28 pixel images of size-normalized hand written digits. Its two partitions include a training set of 60,000 labelled patterns and a test set of 10,000 non-labelled patterns. Approximately 250 writers were used to develop the training set [8]. Since 1998 it has been used as a benchmark data set for a wide variety of machine learning algorithms.

The nature of the problem associated with the MNIST data set is one of classification. The goal is to train a model to classify unseen test images as 1 of 10 classes, the digits contained in the interval [0, 9].

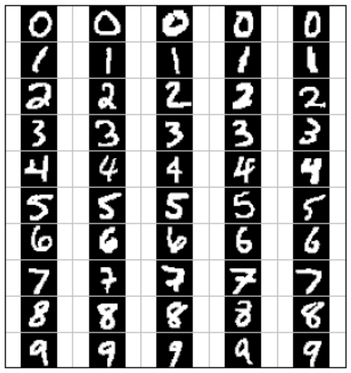

Figure 1: Example of MNIST data patterns. A.Burratin, http://andrea.burattin.net/stuff/handwritten-digit-recognition/

Figure 1 provides a visual representation of what images in the data set look like. Notice differences in pattern across each class. Lacking uniformity, variations between images are a result of differences in writing style, boldness, blurriness and pixelation. These characteristics are referred to as noise in data. Such noise presents levels of difficulty in training models to classify unseen digits. The following section describes the model used to sort through this noise and classify handwritten digits.

3. Multiclass Perceptron

The multiclass perceptron is a manifestation of the binary perceptron model Frank Rosenblatt proposed in 1958 [10]. The purpose of a perceptron is to classify data points into respective categories. Its multiclass extension generalizes for multiple class problems [11]. As a result, the multiclass perceptron proves a natural fit for the MNIST data set.

This model is composed of three parts. Multinomial logistic regression (MLR) acts as an activation function while the negative log likelihood measures its *cost*. Finally, the parameters of our model are updated iteratively using stochastic gradient descent.

3.1 Multinomial Logistic Regression

Data which can be divided into two classes is typically separated using a linear equation.

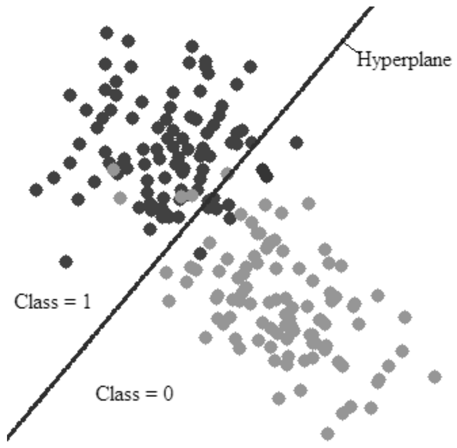

Figure 2: Illustrates linear classification on data that is binary in nature.

http://sli.ics.uci.edu/Classes-CS178-Notes/LinearClassify

In figure 2 we see a single linear function separating two classes of data. This line is called the *hyperplane*.

The logistic regression function emulates this behaviour while measuring the probability of its assignments matching labels of data. This sigmoid activation function produces a value between zero and one with 0.5 representing the case where a data point resides on the differentiating *hyperplane*. If the value produced is greater than 0.5 the neuron is considered active. The opposite is true for values less than 0.5.

The logistic regression model is defined as

$$P(Y = y \mid X = \mathbf{x}_i, \mathbf{w}) = \frac{1}{1 + e^{W_i x + b_i}} \quad (1)$$

where W and b are weight and bias vectors respectively. The equation produces a number between zero and one with 0.5 being the threshold, hence, its binary nature.

It helps to think about this process using a concrete example. Following a metaphor used in "Learning From Data: A Short Course", consider the task of credit approval. Many factors are accounted for when assessing an individual's ability to pay off debt. These factors are represented by x_i in our model. The weights in W and biases in b associated with each x are adjusted throughout training based on historically labelled data. These adjustments affect the denominator's linear expression which dictates the probability output. When an output is greater than 0.5 the neuron is activated and we accept an individuals credit request. When below 0.5 we deny them credit [11].

The classification problem associated with MNIST demands a model where an output can take on 1 of 10 classes. Conveniently, the logistic regression function has been proven to generalize well for multiclass problems. The commonly used MLR model is defined as

$$\begin{aligned} P(Y = i \mid x, W, b) &= softmax_i(Wx + b) \\ &= \frac{e^{W_i x + b_i}}{\sum_j e^{W_j x + b_j}} \end{aligned} \quad (2)$$

for $i \in \{0, 1, 2, \ldots, 9\}$, where $W_{i\&j}$ are weight matrices and $b_{i\&j}$ are bias vectors [12]. The algorithm measures the probability of a matching output for each class in a vector where the sum of probabilities is one. The maximum probability in the vector is chosen as the single output for the neuron. The class, or in our case, the digit associated with the chosen index represents the activated class. For instance, if the output vector index with the greatest probability was one, we'd classify the digit as a 1.

3.2 Negative Log Likelihood

To maximize the probability of our model representing labelled data we use a loss function which quantifies the deviation of our model from the labelled data in our training set. We call this deviation the "cost" of the current model and seek to maximize the likelihood of proper representation by minimizing the cost [12].

We employ the commonly used negative log likelihood as our loss function. For MLR it is defined as

$$\mathcal{L}(\theta = \{W, b\}, \mathcal{D}) = \sum_{i=0}^{|\mathcal{D}|} \log(P(Y = y^{(i)} | x^{(i)}, W, b)) \quad (3)$$

where W is the weight matrix, b is the bias vector and *D* is the magnitude or count of the entire data set.

In our implementation, the mean of costs across a batch is used in lieu of the sum of costs. In section four, we'll see this is done for consistency in descending our model.

3.3 Mini-Batch Stochastic Gradient Descent

Mini-batch Stochastic gradient descent (MSGD) is an algorithm for minimizing the cost computed by our loss function. Descending the loss function and updating model parameters is accomplished by subtracting from each parameter, the derivative of the loss function with respect to W and b. It is formally defined as

$$\begin{aligned} w_j &= w_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (A_{w,b}(x^{(i)}) - y^{(i)}) x_j \\ b &= b - \alpha \frac{1}{m} \sum_{i=1}^{m} (A_{w,b}(x^{(i)}) - y^{(i)}). \end{aligned} \quad (4)$$

where A is the activation function with weights, W_k, biases, b, and m being the batch size [7][14].

Our implementation uses a sampling technique called mini-batching which reduces the sample space used for updating to a magnitude less than m. This design decision alone provides algorithmic value to our training process.

Executing MSGD descends the model till the optimal minimum is reached. When this convergence occurs, we consider the model trained. In this way, we facilitate computational learning. Figure three illustrates this process nicely.

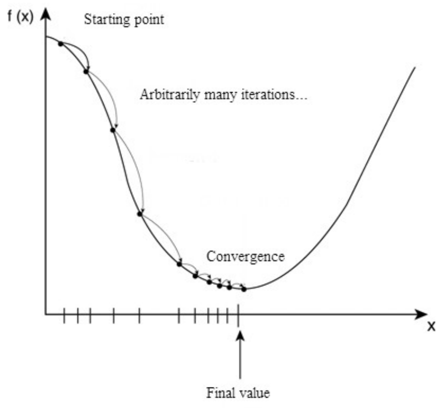

Figure 3: Stochastic gradient descent.
http://www.yaldex.com/game-development/

3.4 Network Overview

Perceptrons are meant to classify data into arbitrarily many classes. A binary perceptron differentiates between data of two types. A multiclass perceptron generalizes differentiation to arbitrarily many classes. These ANNs are composed of many cohesive neurons.

In figure four we provide a visual representation of the binary neuron present in the normal perceptron. From left to right we see a sigmoid function taking arbitrarily many inputs. In the case of credit approval, these inputs could be metrics such as an individual's annual salary, years in residence, outstanding loans, etc. A ratio, $y_{i,}$ is calculated by the sigmoid function. If the 0.5 threshold is reached, the neuron is activated and we accept the credit request. If not, credit is denied.

Now, consider again our case where classification across multiple categories is necessary. The multiclass neuron using MLR is shown in figure five no the next page.

Notice the series of sigmoid functions at the nucleus layer of the neuron. Each contributes a probability for our ten classes from which the maximum probability is chosen as the output. In our case, 7 nodes would need to be added to the figure for it to be fully representative of our network.

4. Optimization and Results

A prominent goal within ML is to train networks faster. Optimization techniques are grouped into two different categories. Algorithmic advantage results from tuning a network's underlying model while computing advantage is attained by leveraging more efficient hardware.

We aim to increase the performance rate of our model by adjusting parameters and leveraging the GPU. To get a sense of results for each training iteration we must be able to quantify performance. This metric is defined as a rate calculated as

$$\frac{E}{(Te - Ts)} \quad (5)$$

where E is the number of *epochs* or passes over the entire training set and $T_e - T_s$ denotes the real time spent training the model [7][13]. For example: if it takes our model 70 epochs and 10 seconds to sufficiently train, our rate of

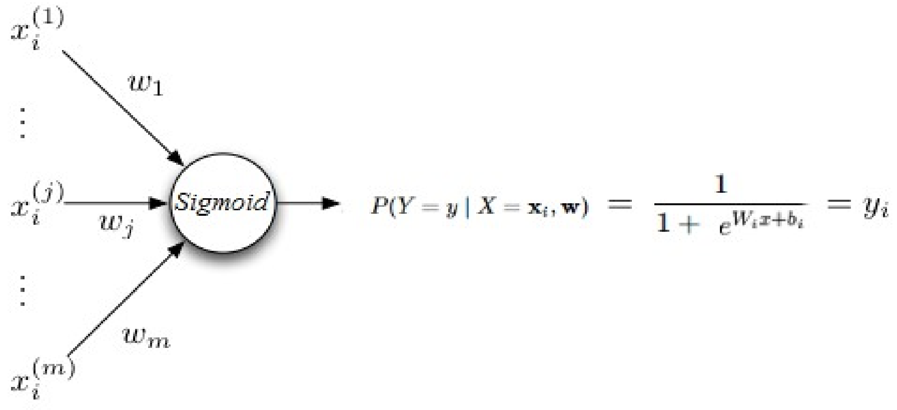
$x_i^{(1)}$
w_1
$x_i^{(j)}$
w_j
Sigmoid
$x_i^{(m)}$
w_m
$P(Y = y \mid X = \mathbf{x}_i, \mathbf{w}) = \frac{1}{1 + e^{W_i x + b_i}} = y_i$

Figure 4: Binary perceptron neuron.

http://briandolhansky.com/blog/2013/7/11/artificial-neural-networks-linear-classification-part-2

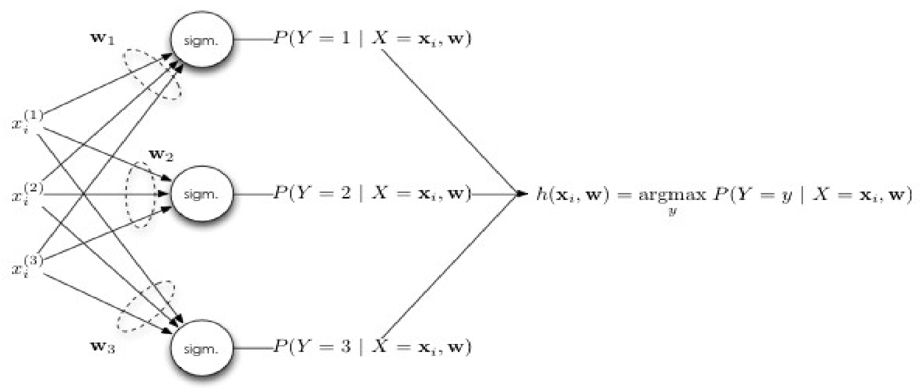
$\mathbf{w}_1$
$\mathbf{w}_2$
$\mathbf{w}_3$
$x_i^{(1)}$
$x_i^{(2)}$
$x_i^{(3)}$
sigm.
sigm.
sigm.
$P(Y=1 \mid X=\mathbf{x}_i,\mathbf{w})$
$P(Y=2 \mid X=\mathbf{x}_i,\mathbf{w})$
$P(Y=3 \mid X=\mathbf{x}_i,\mathbf{w})$
$h(\mathbf{x}_i,\mathbf{w}) = \operatorname{argmax}_y P(Y=y \mid X=\mathbf{x}_i,\mathbf{w})$

Figure 5: Multiclass perceptron neuron.

http://briandolhansky.com/blog/2013/9/23/artificial-neural-nets-linear-multiclass-part-3

performance would be seven *epochs* per second (EPS).

It's important effectiveness remains stable when implementing optimization strategies. We quantify effectiveness by calculating an error percentage over an unlabelled test set. For each digit in our test set the model's output is compared to the digit's actual value. The summation of misclassified digits is then divided by the magnitude of the test set. The result is a percentage representing our model's effectiveness.

Our implementation achieves a minimum test error rate of 7.073 percent with the top performing configuration yielding a GPU performance rate of 13.213 EPS.

4.1 Parameter Tuning

The performance of our model can be governed by the batch size and learning rate parameters of the MSGD algorithm. To test their effect, we train the model using different batch size-learning rate configurations. In total the model was trained 2490 times with performance rates ranging from 1.497 EPS on a CPU to 13.213 EPS on a GPU. The remainder of this sub-section will use GPU statistics for clarity and consistency in reviewing results.

4.1.1 Learning Rate

The *learning rate,* denoted as $\propto$ in equation four, is the step size for our SGD algorithm. Smaller values of $\propto$ translate to smaller steps during gradient descent resulting in longer training times. The opposite is true for larger values of $\propto$. During experimentation, we keep the learning rate constant for each training sequence to more accurately measure its affect on performance.

Figure three displays varying step sizes along a graphed function. The algorithm used to produce this figure used more sophisticated techniques for adapting the learning rate during the training process. Still, it clearly illustrates how varying learning rates result in different step sizes during gradient descent.

Since the length of time is the denominator of our performance rate calculation, we expected larger values of $\propto$ to produce shorter times and larger performance rates. Our generated data showed this intuition to be incorrect. Figure six displays performance rates based on learning rates per batch size. Notice each line maintains a constant overall trend suggesting that our model elicits no relationship between the learning rate and performance.

Hindsight research leads us to suspect this was due to the averaging done over the cost function. Since the data set is never shuffled and the math from training sequence to training sequence remains the same, taking the average, rather than the sum of equation three, effectively normalizes the stepping behaviour across batch tests.

4.1.2 Batch Size

The *batch size,* denoted as m in equation four defines the sample size on which we run the MSGD model. From the start, using this form of stochastic gradient descent provides algorithmic gains during the training process. Instead of updating weights at each point over the entire data set (as is done in SGD), we update weights

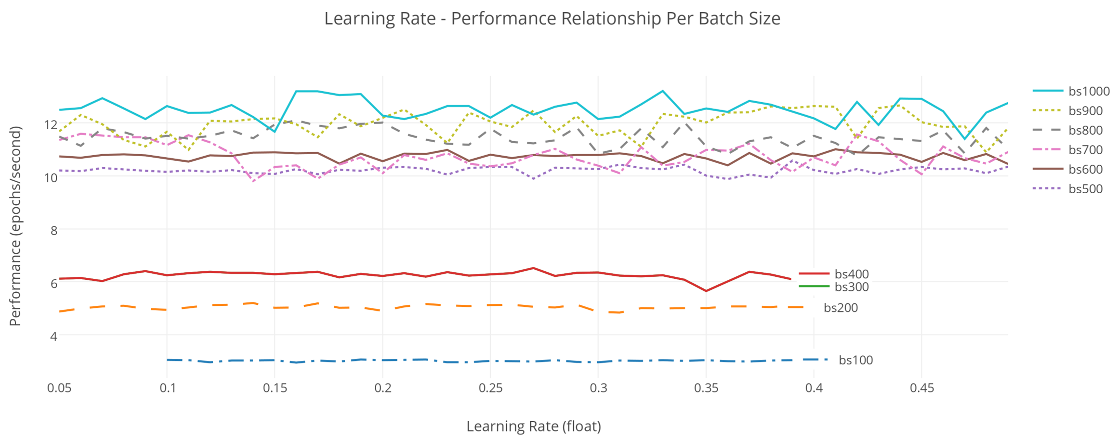

Figure 5: Shows the relationship between learning rate and performance per batch size.

Note: To visually differentiate learning rate sequences per batch size, we use color within and line style variations within this figure.

Generated using plot.ly.

over sporadic partitions within the set. The effect is a drastic reduction in the number of gradient computations performed.

Increasing the mini-batch size typically decreases a model's rate of convergence since greater numbers of gradient computations are needed [15]. This results in a greater number of epochs. On the other hand, opportunities for parallelization increase with greater mini-batch sizes[14]. When parallelilized, the machine is able to process gradient computations at a faster rate. This relationship is shown in figure six. As the batch size increases, so does the performance rate for the GPU and CPU.

Despite increasing the overall time it takes to converge, the rate of passes over the data or the *epoch* rate increases.

Notice the plateauing behaviour of both curves. We can expect this trend to continue because at some point the increasing convergence rate would outweigh the performance gains attained through parallelization. Tuning parameters is an act of trial and error.

4.2 GPU Utilization

In 4.1.2 we saw that increasing the mini-batch size gave opportunity for parallelization of gradient computations on both machines. This parallelization is especially effective on the GPU where the number of cores is exponentially

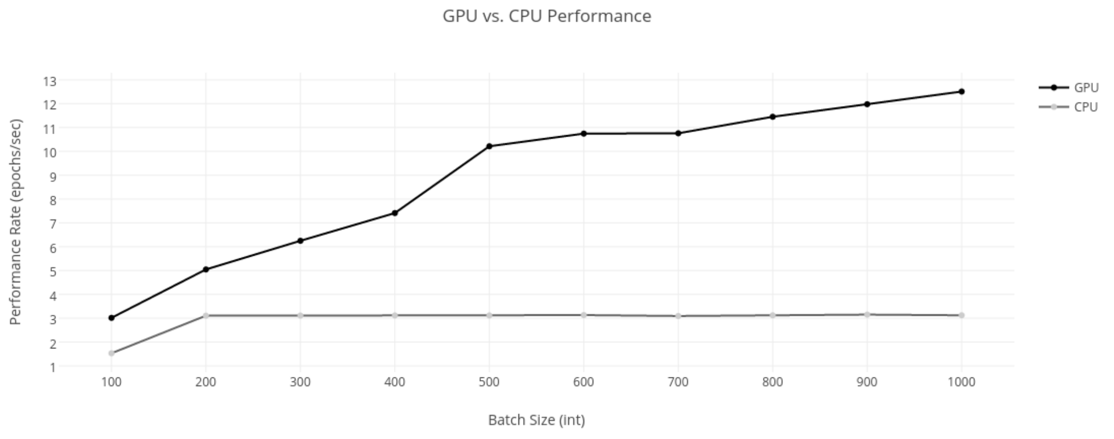

Figure 6: Illustrates the increase in GPU performance with respect to the batch size while contrasting performance rates shown by the CPU.

Generated using plot.ly.

greater than on the CPU. Parallelization takes place on a much greater scale. Figure six illustrates this relationship.

We can conclude from our experimentation that leveraging Theano's use of the GPU can increase performance rates by 4.286X that of the CPU.

5. Conclusion

Given the advanced state of the ML field, this paper provided an assessment of baseline methods for optimizing ANNs. Specifically it focused on tuning the learning rate and batch size parameters of the MSGD algorithm on both the CPU and GPU. Training was done on the MNIST data set using a test set of 60,000 hand written digits. We generated data by running our model 2490 times using distinct learning rate-batch size configurations.

Our experimentation showed that GPU utilization is key to optimization in systems for training ANNs. The GPU's ability to parallelilize multiplication in dimensions greater than one produce significant performance gains over the CPU. For us, using the GPU yielded a maximum performance increase of 4.286X that of the CPU.

We had hypothesized that performance rates would increase with the size of the learning rate since this parameter helps dictate the size of step during MSGD. After generating data, we found our implementation did not present this relationship. We suspect this was a result of step size normalization through averaging costs. In the future, we wish to continue experimentation to validate our assumption.

Increasing the batch size up to a batch size of 1000 increased performance due to parallelilization opportunities. The plateauing behaviour observed suggests further training sequences would be necessary to measure the upper limit of performance attainable by our model and system.

There exist more sophisticated algorithms for adapting the learning rate throughout training and for increasing mini-batch size without decreasing convergence rates. These algorithmic designs are of particular interest for future research.

Despite deep convolutional neural networks (CNN/DNN), the field's newest family of neural networks, yielding state-of-the-art performance on MNIST achieving error rates as low as 0.23 percent [9], the simplicity of our implementation provided clarity with empirical measurements.

Finally, we hope that our observations serve as a baseline guide for developing intuition behind novice network training!

6. Acknowledgements(optional...)

References:

[1] W. McCulloch, W. Pitts, A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics, vol. 5*, 1943, p. 115-133.

[2]A. Coates, B Huval, T. Wang, D. Wu, A. Ng, B. Cantanzaro, *Deep learning with COTS HPC systems. Journal of Machine Learning Research, vol. 28(3)*, 2013, p. 1319-1327.

[3] A. Trask, D. Gilmore, M. Russell, Modeling order in neural word embeddings at scale, *ArXiv preprint arXiv:1506.0238*, 2015.

[4] B. Krishnapuram, L.Carin, Sparse Multinomial Logistic Regression: fast algorithms and generalization bounds. *Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. 27(6)*, 2005, p. 957-968.

[5] D. Bo¨hning, Multinomial Logistic Regression Algorithm. *Annals of the Inst. of Statistical Math., vol. 44*, 1992, p. 197-200.

[6] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, Y. Bengio. Theano: new features and speed improvements, *Deep Learning and Unsupervised Feature Learning NIPS* Lake Tahoe, CA, 2012 Workshop, 2012.

[7] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Wade-Farley, Y. Bengio. Theano: A CPU and GPU math expression compiler, *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2010.

[8] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 86(11),* 1998, 278-2324.

[9] D. Cireşan, U. Meier, J. Schmidhuber, Multi-column deep neural networks for image classification. *IEEE Conference on Computer*

Vision and Pattern Recognition, Providence, Rhode Island, 2012, 3642-3649.

[10] F. Rosenblatt, The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review, vol. 65(6),* 1958, 386-406.

[11] Y. Abu-Mostafa, M. Magdon-Ismail, H. Lin, *Learning from data: A short course chp. 1,* 2012, p. 1-8

[12] B. Krishnapuram, L. Carin, M. Figueiredo, A. Hartemink, Sparse multinomial logistic regression: fast algorithms and generalization bounds. *IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 27(6).* 2005, 957-968

[13] C. Shalizi, Chapter 11 logistic regression, *Advanced Data Analysis from an Elementary Point of View.* 2016, p.245-274

[14] Y. Bengio, Practical Recommendations for Gradient-Based Training of Deep Architectures V2, *Prepublish – arXiv:1206.5533v2*, 2012

[15] M. Li, T. Zhang, Y. Chang, A. Smola, Efficient mini-batch training for stochastic optimization. *In ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD),* 2014

[16] K. Lee, S. Piantino, Facebook to open-source AI hardware design. *code.facebook.com/posts,* 2015

[17] Theano Development Team, Logistic Regression, *Deep Learning Tutorials.* 2008-2013