A comparison of iterative optimizers applied to the MNIST dataset

Suhrid Deshmukh, Ismail Degani

6.337 Final Project
May, 6th 2017

Abstract

Abstract for algorithms used and performance on the bench mark problem.

1 Introduction

Machine learning and optimization has found home in some of the most important machine learning(ML) applications. The fields of machine learning and mathematical programming are becoming increasingly intertwined[3]. Optimization problems lie at the heart of most machine learning approaches. Many optimization algorithms that are applied in machine learning problems converge at different rates. Many of the learning algorithms also have different memory footprint and different computational complexity. Studying how different optimization algorithms work when applied in ML applications is therefore an interesting field of research.

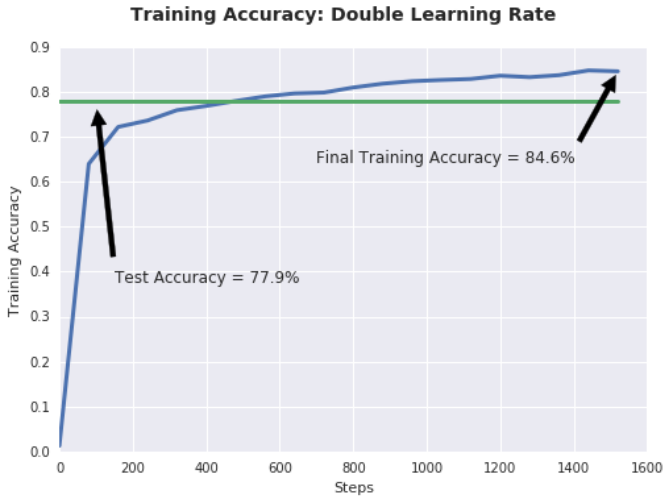
The work here explores different optimization schemes that are used in training weight matrices in a Neural net that uses logistic regression. The learning problem in neural nets is formulated in terms of the minimization of a loss function. We can combine the weights and the biases in a neural net in the loss function weight vector w. The loss function is then f(w) with w* as its minima. The learning problem in neural nets is basically searching for a weight vector w* such that the value of the loss function f(w) is minimum at w*. This means that if we evaluate the value of the gradient of f(w) at w*, the gradient is zero.

The loss function in general is a multi-dimensional nonlinear function. In order to minimize the loss function, typically iterative optimization methods are used. The iterative algorithm calculates the loss at each iteration, the change of loss between two steps is called loss decrement. The training algorithm stops when the specified criteria is met. Usually the training algorithm stops when the loss decrement falls below a particular threshold. There are multiple training algorithms that are available to train a neural net. We consider three different algorithms to train a neural net; Gradient descent, Quasi Newton (BFGS),Adam-Optimizer.

The algorithms mentioned above are applied on a simple problem first to make sure that they are functioning properly. They are then used on a benchmark problem which is the classification of handwritten digits in a MNIST data set using a neural net that uses logistic regression. The next section describes each of the algorithms in detail along with the pseudocode and their performance characteristics.

2 Algorithms

2.1 Gradient Descent



The gradient descent method is a natural development of Laplace's method applied to the asymptotic estimate of integrals of analytic functions [4]. Mathematicians have often attributed the method of steepest descent to the physicist Peter Debye, who in 1909 worked it out in an asymptotic study of Bessel functions. Gradient descent method is a way to find a local minimum of a function[4]. The way it works is it starts with an initial guess of the solution and then it takes the gradient of the function at that point. We step the solution in the negative direction of the gradient and we repeat the process. The algorithm will eventually converge where

the gradient is zero (which correspond to a local minimum). Gradient descent algorithm requires us to choose a learning rate, but it works well to solve the normal equation even when the system is very large. Direct methods might take a long time to solve for the solution, but iterative methods like gradient descent, combined with line search can reach the solution in very few iterations.

Gradient descent is based on the idea that for a multi-dimensional function, if the derivative exists in the neighborhood of a point, then the function decreases the fastest in the negative direction of the gradient of the function at that point. This results in the algorithm starting from an initial guess and then repeatedly calculating the gradient at the new guess and moving in the direction opposite to that gradient. The next section describes the pseudocode for gradient descent.

2.1.1 Gradient Descent Pseudocode

Inputs: (P, V, η, ϵ) where P is the parameter space, $V : P \to R$ is the objective function, η is the step size, and ϵ is the tolerance for the stopping threshold.
Output: A point $p^* \in P$ that approximates the minimum of the function V.
Process:

1. Randomly select an initial guess p_0 in the parameter space P
2. Let $i \leftarrow 0$
3. Repeat until $\delta V < \epsilon$
   (a) Compute the gradient vector $\nabla V(p_i)$ of V at p_i if possible or estimate $\nabla V(p_i)$ as the direction in which V increases fastest from p_i
   (b) Let $p_{i+1} \leftarrow p_i - \eta \nabla V$
   (c) Let $\delta V \leftarrow V(p_i) - V(p_{i+1})$
   (d) Let $i \leftarrow i + 1$
4. Output p_i

2.2 Adam Optimizer

The Adam Optimizer[2] is a member of a class of "adaptive" gradient descent optimizers which adjust their learning rate α at each iteration in order to converge more efficiently. This adaptive behavior significantly reduces the impact of a badly chosen initial learning rate α, which can be difficult to determine. Adam is specifically optimized for sparse datasets as are many similar algorithms in this class. This makes it an excellent candidate for MNIST data which is monochrome and therefore predominantly zero-value.

The Adam algorithm stores a history of previous gradients v_t and squared gradients m_t, also known as first and second moments. In order to compute the current gradient, Adam applies exponentially decaying factors β_1 and β_2 to the first and second moments. It finally calculates bias-adjusted values of these moments $\hat{v}_t, \hat{m}_t$, and applies a smoothing term ϵ which avoids division by zero. Pseudocode is given below:

2.2.1 Algorithm 1: Adam optimization step

1. while $f(\theta_t) > \tau$
   (a) $t = t + 1$
   (b) $g_t = \nabla_\theta f_t(\theta_{t-1})$
   (c) $m_t = \beta \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
   (d) $v_t = \beta_2 \cdot v_{t-1} + (1 + \beta_2) \cdot g_t^2$
   (e) $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$
   (f) $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$
   (g) $\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}t}+\epsilon}$
2. end while

2.3 BFGS

Some of the optimization algorithms like the Newton's method requires the computation of the Hession or the Jacobian. In case the Hessian or Jacobian is absent or too expensive to calculate, we can resort to quasi-newton methods. Quasi-Newton methods are also used to to calculate the roots of a polynomial or as an optimization algorithm. They are typically used in lieu of Newtons method. One of the popular Quasi-Newton method is the BFGS algorithm[1].This will be the main algorithm that will be explored in detail in this piece of work. The goal of this paper is not to prove that BFGS is the most appropriate algorithms to be used in machine learning problems, but to explore the algorithms on some of the traditional machine learning problems and explore how they perform compared to some competing algorithms.

BFGS essentially is therefore an approximation to the Newtons method. The main advantage of the BFGS method is that you need not evaluate the Hessian explicitly. Newtons method or BFGS are not guaranteed to converge if the function does not have a quadratic Taylor expansion near the optimal point. Also, in BFGS when the Hessian is getting approximated, if the initial guess of the Hessian is a positive definite matrix, the next update of the Hessian is also positive definite. This defines a recurrence relationship between the successive Hessian updates. Also the Hessian is not explicitly constructed in the algorithm but is is constructed using the difference in the input vector $s^{(k)}$ and the

difference in the gradient vector, $y^{(k)}$. The pseudocode for the algorithm in given in the next section.

2.3.1 BFGS pseudocode

When the objective function is quadratic and of the form $f(w) = \frac{1}{2}w^T Qw + w^T b$, the BFGS method with exact line search is given as follows, given an initial point $w^{(0)}$, an initial estimate of the inverse Hessian $H^{(0)}$, and a stopping threshold ϵ.The pseudocode for BFGS is given below.

1. k = 0
2. $\nabla f^{(0)} = Qw^{(0)} - b$
3. Loop
   - (i) $\Delta w^{(k)} = -H^{(k)} \nabla f^{(k)}$
   - (ii) $\alpha^{(k)} = -\frac{\Delta (w^{(k)})^T \nabla f^{(k)}}{\Delta (w^{(k)})^T Q \nabla f^{(k)}}$
   - (iii) $w^{(k+1)} = w^{(k)} + \alpha^{(k)} \Delta w^{(k)}$
   - (iv) $\nabla f^{(k+1)} = Qw^{(k+1)} - b$
   - (v) If $\|\nabla f^{(k+1)}\|_2 \leq \epsilon$ then terminate
   - (vi) $s^{(k)} = w^{(k+1)} - w^{(k)}$
   - (vii) $y^{(k)} = \nabla f^{(k+1)} - \nabla f^{(k)}$
   - (viii) $\rho^{(k)} = \frac{1}{(y^{(k)})^T s^{(k)}}$
   - (ix) $H^{(k+1)} = (I - \rho^{(k)} s^{(k)} (y^{(k)})^T) H^{(k)} (I - \rho^{(k)} y^{(k)} (s^{(k)})^T) + \rho^{(k)} s^{(k)} (s^{(k)})^T$
   - (x) k = k+1
4. return $w^{(k)}$

3 Toy Problem Description

In order to make sure that the implemented algorithms were optimizing to the expected value, all the algorithms were tested on a relatively easy problem with a known solution. The problem that was solved was of the simple format Ax = b. We tested the algorithms with a 3X3 matrix A and a 3 by 1 matrix b. The problem was solved for x. Each of the three algorithms were implemented on this problem and they all converged to the same solution although at different speeds. The toy problem which was solved for x was of the form:

$$\begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

The exact toy problem was with the solution substituted was,

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & -1 & 0 \\ 3 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix} = \begin{bmatrix} 9 \\ 5 \\ 3 \end{bmatrix}$$

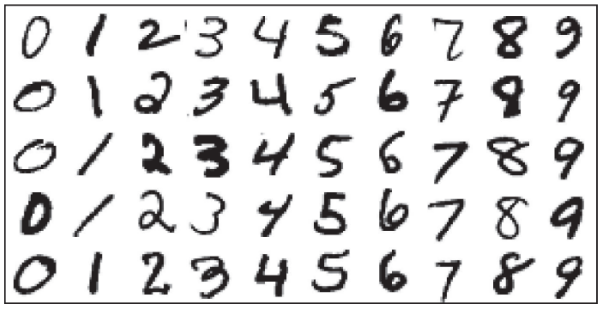


Figure 1. MNIST Dataset

4 Benchmark problem description

Describe your benchmark problems here. Describe the inputs and outputs of the problems such that the reader outside your your field can understand their meanings. If there are numerical challenges associated with these problems, describe them.

There are many traditional ML problems that have been used to test the algorithms on. Modified National institute of standards and technology (MNIST) is one such data set. This is the defacto "hello world" data set for machine learning problems. Since its release in 1999, this classic dataset of handwritten images has served as the basis for benchmarking classification algorithms. As new machine learning techniques emerge, this data set remains a classic dataset to test the techniques on.

The dataset consists of 60,000 training and 10,000 test examples of handwritten digits from 0-9. Various preprocessing is performed on the data to increase contrast and reduce noise. Additional preconditioning involves downsampling to a pixel size of 28*28 to reduce the dimensionality of the problem. Fig 2 illustrates a sample of the dataset:

and the problem here is to ask the machine to classify them from 0-9. Typically Neural nets are used to solve this classification problem. As mentioned in the beginning, Neural nets involve an optimization problem that needs to decide the best weights to use in the neural nets. In order to make the data set more manageable for optimization, some preconditioning was done to make the data set suitable for the algorithm to take as input. Some of the preconditioning involves increasing the contrast in the images and the data set is also down sampled to a 28*28 to reduce the dimensionality of the problem.

The input data is reshaped into a single vector of pixel values. This representation is convenient as it allows the dataset.

MNIST with vizualization Preconditioning of the data set Reduce the dimensionality, reduce the greyscale to 0,1 Downsample from 300dpi to 28*28 Some discussion on logistic regression maxes out at 94% accuracy. Neural nets

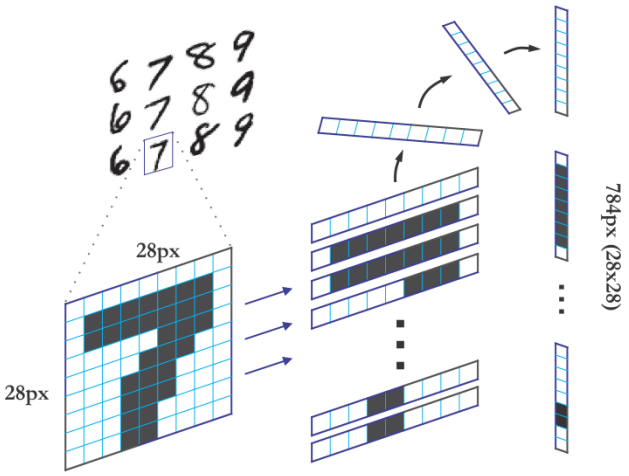


Figure 2. Reshaping input character matrix

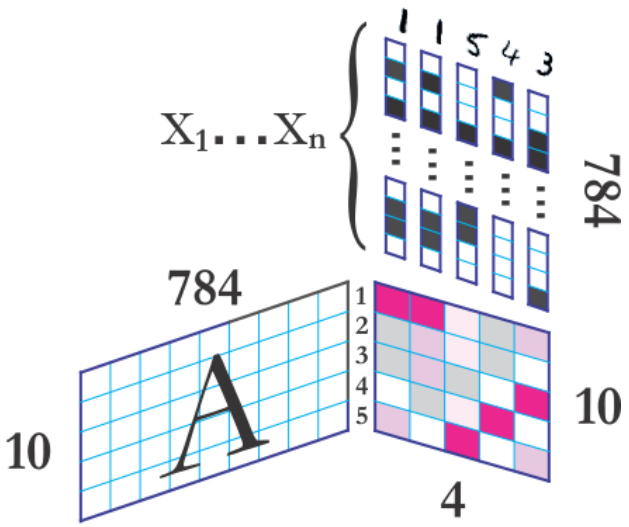


Figure 3. Logistic Regession Overview

improve that to 99.8%.

5 Performance Characteristics

Introduce the performance characteristics you are going to measure. If you have theoretical estimates on these performance characteristics, derive them here.

Some of the performance characteristics that were looked at were the time taken in seconds for the algorithm to run until convergence and the accuracy rate of classification. Comparing the number of flops, theoretical Vs. practical did not make much sense because it was very hardware dependent and some of the packaged implementations were far more optimized that the basic implementations of the otherwise more optimal algorithms. Therefore the trends in time taken for the algorithms to converge to the minimum were compared for different algorithms. The time taken to converge was tested on the example problem as well as the benchmark problem.

Apart from the time taken to convergence another measure of performance was the accuracy with with the algorithms were able to optimize a and perform on the classification problem of recognizing digits and putting them into a class. The accuracy with which the MNIST data set was classified by the three optimization routines was plotted for the three different algorithms to make sure that all the algorithms performed equally well from the accuracy point of view.

5.1 Performance characteristics on example problem of all three algorithms

5.2 MNIST Problem: Gradient Descent Performance Characteristics

5.3 MNIST Problem: Adam optimizer performance characteristics

¡¡¡¡¡¡¡ Updated upstream

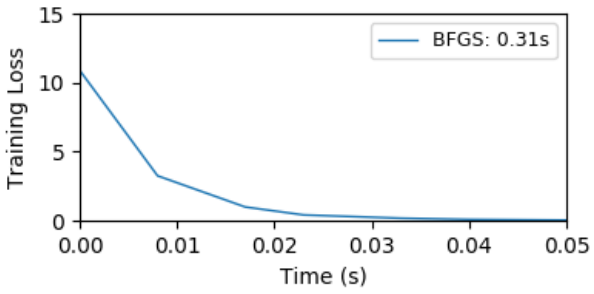


Figure 4. Reshaping input character matrix

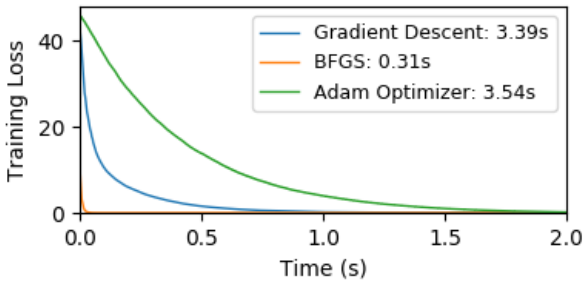


Figure 5. Reshaping input character matrix

5.4 Gradient Descent Performance Characteristics

5.4.1 Gradient Descent Performance Characteristics Toy Problem

5.4.2 Gradient Descent Performance Characteristics MNIST

5.5 Adam Optimizer Performance Characteristics

5.5.1 Adam Optimizer Performance Characteristics Toy Problem

5.5.2 Adam Optimizer Performance Characteristics MNIST

======= ¿¿¿¿¿¿¿ Stashed changes The following section compares gradient descent, adam, and BFGS optmizers on a linear least squares Ax = b optimization, and a logistic regression to solve the MNIST problem.

5.6 MNIST Problem: BFGS Performance Characteristics

6 Numerical results and discussion

Report the numerical results generated by yourself for the benchmark problems. Include numerical measurements of the performance characteristics and compare them with theoretical estimates. Discuss the difference in these results between the algorithms you tested. Accuracies on different data sets Show the python normalized flops plot here

7 Conclusion

Which algorithm was the fastest. Did the accuracy remain consistent throughout. Which algorithm should be used when

References

[1] J.-F. Bonnans, J. C. Gilbert, C. Lemaréchal, and C. A. Sagastizábal. *Numerical optimization: theoretical and practical aspects*. Springer Science & Business Media, 2006.

[2] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[3] A. Munoz. Machine learning and optimization. *URL: https://www. cims. nyu. edu/˜ munoz/files/ml_optimization. pdf [accessed 2016-03-02][WebCite Cache ID 6fiLfZvnG]*, 2014.

[4] S. S. Petrova and A. D. Solov'ev. The origin of the method of steepest descent. *Historia Mathematica*, 24(4):361–375, 1997.