SGDBabysitter

Joshua Catalano | 28675650
Puranjay Rohan Gulati | 96579164

December 21, 2018

Abstract

We created a stochastic gradient descent algorithm that dynamically adjusts learning rate and batch size. To make decisions, the algorithm considers the change in validation error over time and the angle between gradients. We're hoping to exploit our knowledge of the behavior of stochastic gradient descent in order to create an implementation that attains lower validation errors. We consider our algorithm a success if it can attain a lower validation error than the best constant learning rate or learning rates determined by a predetermined series of numbers (e.g. $\frac{1}{t}$ or $\frac{1}{\sqrt{t}}$). In order to test our algorithm, we use it to fit a basic neural network to various real and generated datasets.

Introduction

As we learned in class, Stochastic Gradient Descent (SGD) is a popular optimization algorithm in Machine Learning that allows for fitting various machine learning models to massive datasets. It can be used to fit logistic regression, ordinary least squares, neural networks, and more. While SGD is less computationally costly than gradient descent, it demonstrates unsavory behavior due to the fact that it assesses the gradient using a random subsample of the data. We constructed our own unique implementation that accounts for this strange behavior in the hopes of improving the performance of models that are fit using SGD.

Making a more accurate implementation of stochastic gradient descent is important because it could be used to fit more accurate machine learning models. Since many academic disciplines and the private sector use SGD to fit machine learning models, a more accurate implementation could improve the performance of prediction in all types of applications. This implementation also has the benefit that it does not require 'babysitting' to obtain low validation errors - resulting in less investment of time and knowledge from the end user.

Related Work

Related work (3+ papers working on similar stuff) goes here.

Theory

In order to create this algorithm, we utilized the concepts we learned in class. In this section, we will discuss our understanding of the theory.

When far away from the minimizer, stochastic gradient descent, with an appropriate learning rate, moves us closer to the minimizer with high probability. As it gets closer to the minimizer, the angles between the gradients assessed at all the different values of X become wider and wider. This increases the probability that the next iteration will actually move away from the minimizer. In class, we called this phenomenon the “region of confusion."

By decreasing the learning rate as we enter this region, we can potentially restrict how much the region of confusion pushes us away from the true minimizer. Starting with a very tiny learning rate, however, can make the algorithm take far too long to converge. On the other hand, having too large of a learning rate can give you divergent results. Thus, ideally the learning rate would be large in the beginning but just small enough so that the algorithm doesn't diverge. Then as we approach the minimizer, this learning rate should decay so that we can get closer to the minimizer within the region of confusion.

When selecting batch size, the trade off is between lower computation costs and a more accurate estimate of the gradient. When far away from the minimizer, a small batch size is ideal because the angle between the gradients evaluated at an of the X values should be relatively small, so taking an average of gradients is less beneficial but still increases computation cost. As we approach the minimizer, the angles between these gradients increase, and averaging will be more beneficial as more gradients are likely to be pointing in the wrong direction.

Algorithm

Describe: modifying NN code, building algorithm, logic.

Analysis

To test our algorithm, we're going to be comparing it to a manually optimized vanilla version of SGD on various datasets.

Discussion

We also applied our SGD algorithm and the Neural Network to a problem in econometrics. When estimating parameters using linear regression, economists are frequently interested in interpreting the coefficients as the effect of one

variable on another. Missing data can be a problem for such an interpretation. If data is missing at a random, then there is no issue and you can just run regression on data that is not missing with no fear of bias in your estimated coefficients. On the other hand, if data is systematically missing, excluding it will cause these estimates to be biased. In these circumstances, it may be worth imputing data (i.e. filling in the missing values) with machine learning techniques.

In order to assess the effectiveness of these techniques, we must generate the data ourselves, so we can know the true parameters of what we are investigating and which technique is the most effective. For part of our testing, we generate data X1 and X2 by drawing from a normal distribution. Then we generate X3 by setting it equal to X1 and X2 plus some random unobserved error. We do this in order to establish some causal relationship between X3 and X1/X2. Then we choose a specification for y, the dependent variable of interest. In this circumstance, we chose $y = 2X1 - 3X2 + 4X3 + \epsilon$ where epsilon is some normally distributed error term. In this specific circumstance, the parameters of interest are 2,-3, and 4. The closer we can get to these numbers, the better the machine learning technique is for data imputation. After generating the data, we run OLS on it to make sure everything is working properly, and we get estimates that are very close to [-2 3 4]. Next, I simulate the removal of data by looping through each example and removing approximately 50% of examples that meet certain criteria. We run OLS on the non-missing data to show the bias this introduces into the estimates. We then train a neural network on the data that is not missing using a vanilla SGD implementation and our SGDBabysitter implementation.

We use these trained weights to predict on the missing data. We then fill in the missing values with these predicted values and run OLS on all of the data (both examples that had missing values and those that did not) to obtain two sets of coefficients, one from the vanilla implementation and one from ours. We run this simulation five times and obtain the following relatively positive results:
True Coefficients: [2,-3,4]
All Data Coefficients: [1.99537, -2.99519, 3.99441]
Coefficients with Missing Data:[2.06545, -3.1007, 4.13595]
Coefficients with NN Imputation (SGDBabysitter): [1.99037, -3.08193, 4.11074]
Coefficients with NN Imputation (VanillaSGD): [1.94844, -3.10303, 4.12519]