EECS 284 Final Report – Cost Based Optimization

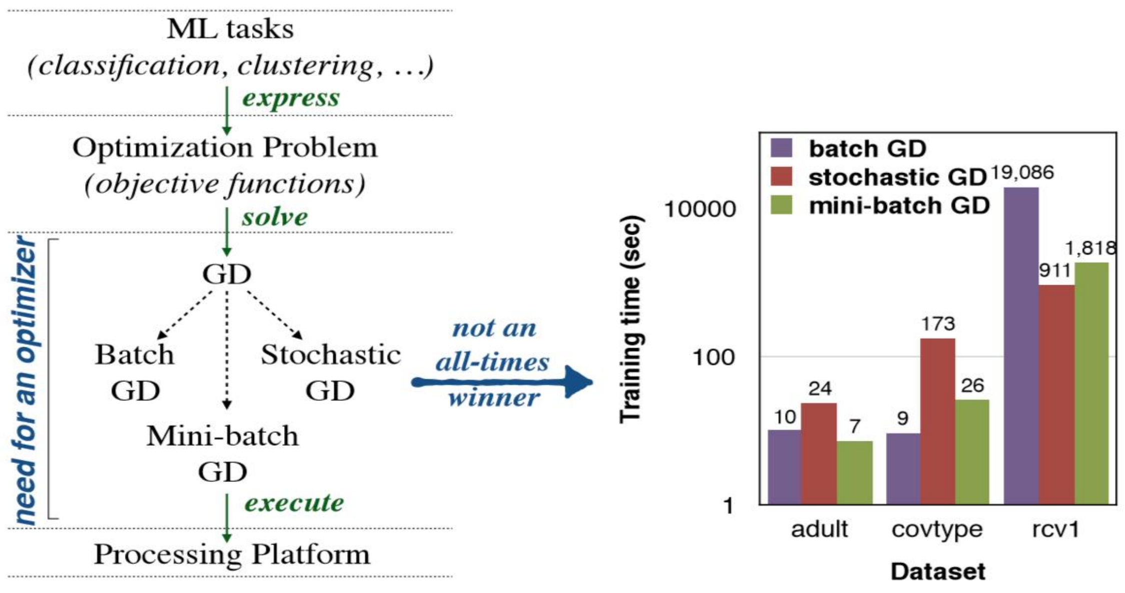
Introduction

Gradient descent (GD) is a highly popular algorithm used to solve various mathematical optimization problems, such as classification. GD can be applied to most machine learning (ML) problems, such as support vector machines (SVM), logistic regression, matrix factorization, conditional random fields, and deep neural networks. There are many different GD algorithms, but three of them are fundamental: batch gradient descent (BGD), mini-batch gradient descent (MGD), stochastic gradient descent (SGD). BGD gives the most accurate results, but it requires a costly full scan over the entire data.

Each algorithm has its advantages and disadvantages, with respect to accuracy and runtime performance. SGD has extremely quick runtime (for gradient computation), since it only considers one example for the gradient computation. However, considering one example for the gradient computation means that the model is learning based off just one example. This creates some randomness in the accuracy of the model. BGD gives the most accurate results, but it requires a costly full scan over the entire data, so the runtime is a bit slow. MGD is in the middle of SGD and BGD in terms of performance. It has a decent runtime, since it considers a small set of examples for gradient computation.

Problem Definition

There are many real-world problems that are solved using various gradient descent (GD) algorithms. As stated earlier, each GD algorithm has their own pros and cons, and each one will perform differently on different models and datasets. The problem is, which GD algorithm will be the most efficient for a specific dataset or model? The diagram below shows that certain gradient descent algorithms out-perform each other (in terms of training time) depending on the dataset or model.



In relational databases, this problem also exists! (For example, what is the best order to compute joins? For those who aren't familiar with join computation, another example would be determining the optimal order of matrix multiplication to reduce computation.) What did we do to solve it? We created an optimizer that considers possible paths of query execution, and it chooses the one with the least amount of cost.

We want to apply this same ideology to ML Systems. Rather than optimizing the order of joins for a database query, we want to optimize the runtime for training a model using gradient descent. Given parameters (runtime, or desired error), the optimizer should choose the optimal GD algorithm to satisfy the requirements.

Preliminaries & Related Work

In the paper, "A Cost-based Optimizer for Gradient Descent Optimization", the researchers designed a cost-based optimizer for ML systems that evaluates the different ways of executing the ML task using a GD algorithm, and chooses the optimal GD algorithm. The main goal of an optimizer is to test different paths of execution and choose the "best" one. In order to know which path is the "best", we need to have some sort of cost function.

In databases, when we are optimizing the order of joins, our cost function would be the total number of rows that will be joined between tables. The researchers of this paper designed their cost function to be a combination of many variables. The table below shows all of the costs that they considered when determining the efficiency of a GD algorithm.

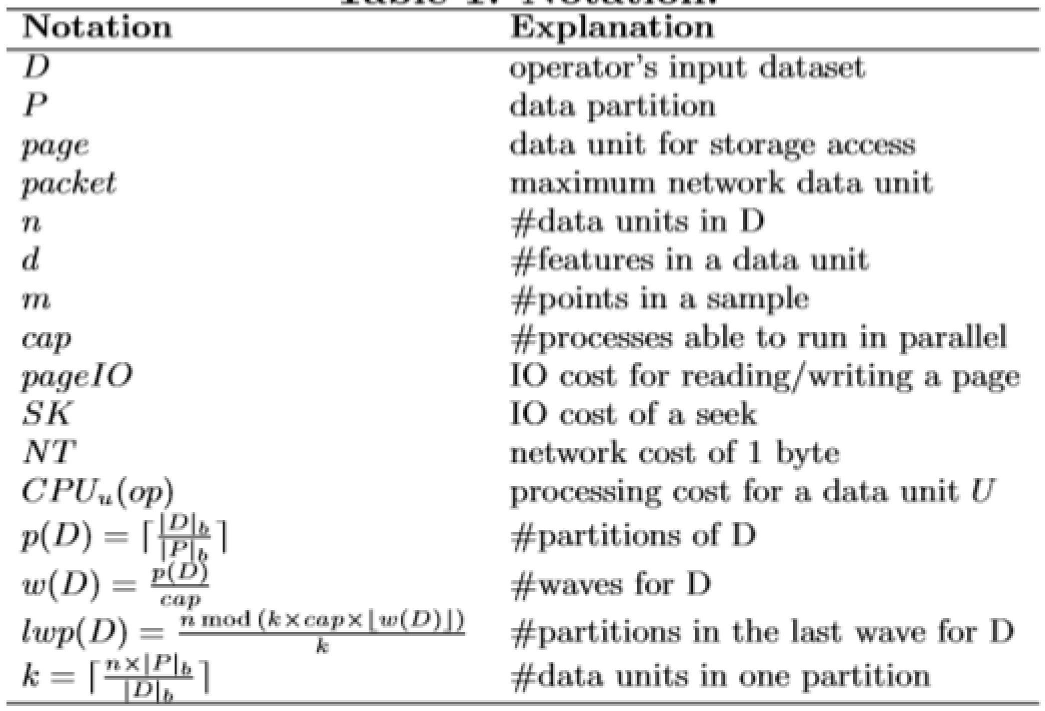


Table 1: Notation.

Notation	Explanation
D	operator's input dataset
P	data partition
$page$	data unit for storage access
$packet$	maximum network data unit
n	#data units in D
d	#features in a data unit
m	#points in a sample
cap	#processes able to run in parallel
$pageIO$	IO cost for reading/writing a page
SK	IO cost of a seek
NT	network cost of 1 byte
$CPU_u(op)$	processing cost for a data unit U
$p(D) = \lceil \frac{\lvert D\rvert_b}{\lvert P\rvert_b} \rceil$	#partitions of D
$w(D) = \frac{p(D)}{cap}$	#waves for D
$lwp(D) = \frac{n \bmod (k \times cap \times \lfloor w(D) \rfloor)}{k}$	#partitions in the last wave for D
$k = \lceil \frac{n \times \lvert P\rvert_b}{\lvert D\rvert_b} \rceil$	#data units in one partition

Putting all these costs together would look something like this (IO cost, CPU cost, Network cost):

$$c_{IO}(D) = \lfloor w(D) \rfloor \times (SK + \frac{|P|_b}{|page|_b} \times pageIO) +$$
$$(SK + \frac{|\min(lwp(D), 1) \times k|_b}{|page|_b} \times pageIO)$$

$$c_{CPU}(D, op) = \lfloor w(D) \rfloor \times k \times CPU_u(op) +$$
$$\lceil \min(lwp(D), 1) \times k \rceil \times CPU_u(op)$$

$$c_{NT}(D) = \frac{|D|_b}{|packet|_b} \times NT$$

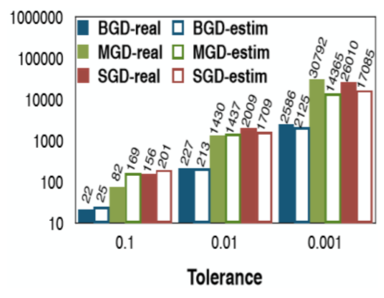
The summation of these costs would give the operator cost:

$$c_{op}(D) = c_{IO}(D) + c_{NT}(D) + c_{CPU}(D, op)$$

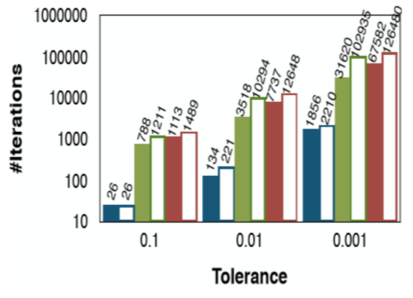
Considering all operators in a specific algorithm would give you the cost for that algorithm (The term "operator" refers to parts of the GD abstraction that this paper had. In my project, my cost function is much simpler):

$$C_{BGD}(D) = c_{\mathcal{S}}(D) + c_{\mathcal{T}}(D) + T \times (c_{\mathcal{C}}(D) + c_{\mathcal{U}}(D) + c_{\mathcal{CV}} + c_{\mathcal{L}})$$

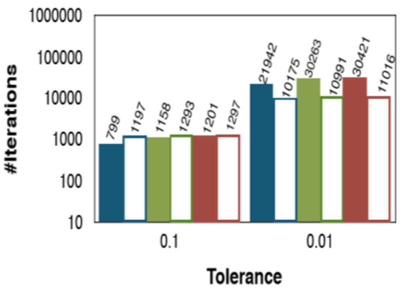
Using this cost function, the researchers were able to successfully choose the optimal GD algorithm. They were also able to accurately estimate the number of iterations and training time for convergence. See the below graphs for results.



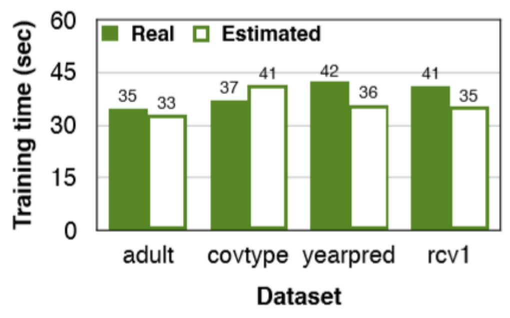
(a) adult dataset



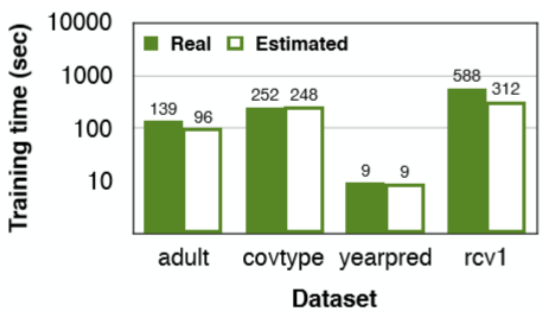
(b) covtype dataset



(c) rcv1 dataset



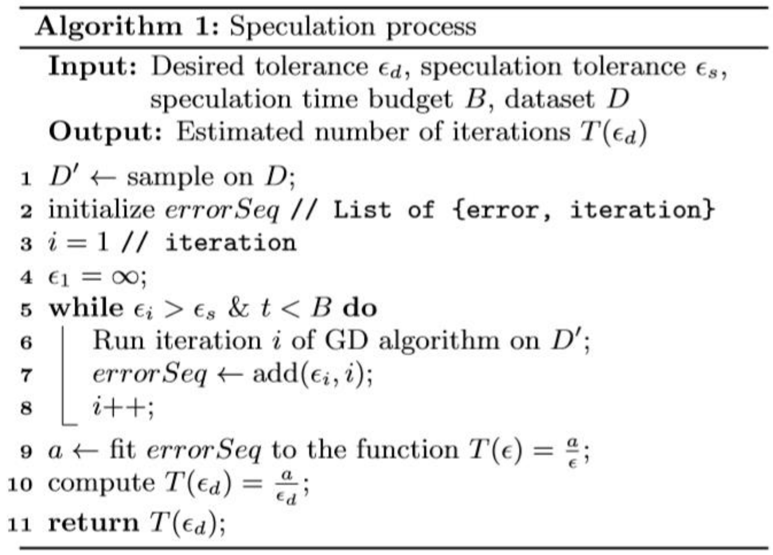
(a) Run of 1,000 iterations



(b) Run to convergence

In my opinion, the most critical part of their cost function would be the iterations estimator (See the pseudocode below). The main steps of this iterations estimator are:

1) Run a GD algorithm on a smaller dataset (For example, run it on 5% of the original dataset).
2) Keep track of the change in error between iterations.
3) Use the all of the change in errors to create an equation, where the input is a desired change in error, and the output is the estimated number of iterations. (Basically take all of your change in errors, and fit it to the curve "a/x", where "a" is some arbitrary constant, and "x" is some change in error.)
4) We want this iterations estimator to finish quickly, since we will be using it multiple times (One for each GD algorithm). The goal of the optimizer is to *quickly* find the most efficient GD algorithm. To ensure that this iterations estimator runs quickly, there are two termination conditions. The first is *reaching the desired change in error*, and the second is *reaching the time budget* allocated to this iterations estimator. If either one of these conditions are met, we stop the iterations estimator.



Algorithm 1: Speculation process

```
Input: Desired tolerance $\epsilon_d$, speculation tolerance $\epsilon_s$,
       speculation time budget $B$, dataset $D$
Output: Estimated number of iterations $T(\epsilon_d)$
$D' \leftarrow$ sample on $D$;
initialize $errorSeq$ // List of {error, iteration}
$i = 1$ // iteration
$\epsilon_1 = \infty$;
while $\epsilon_i > \epsilon_s$ & $t < B$ do
    Run iteration $i$ of GD algorithm on $D'$;
    $errorSeq \leftarrow$ add($\epsilon_i, i$);
    $i$++;
$a \leftarrow$ fit $errorSeq$ to the function $T(\epsilon) = \frac{a}{\epsilon}$;
compute $T(\epsilon_d) = \frac{a}{\epsilon_d}$;
return $T(\epsilon_d)$;
```

Once we have created our estimated iterations equation, we are able to predict the number of iterations needed to reach some certain change in error. We than can take this estimated number of iterations and multiply it to the average time per iteration to get the estimated total runtime.

<u>Technical Contributions (What did I do in the project)</u>

For my final project, I wanted to extend and modify this cost-based optimizer. I started off with two small changes.

1) The first change I wanted to make was the cost function. Rather than having a complicated cost function, I wanted to have a more simplified cost function. The way I decided to measure cost was simply by looking at the estimated runtime for the GD algorithm. Using the pseudocode for the iterations estimator from the research paper, I measure the estimated number of iterations (on a smaller dataset). Then I multiply that

estimated number of iterations with the average runtime for one iteration (on the full dataset). This will give me my estimated total runtime, and this is what I will use to compare GD algorithms with each other.

2) After simplifying the cost function to measure runtime, I decided to also change the equation that we fit our change in errors to. Rather than fitting our change in errors to (a/x), I decided to fit them to the curve (a/x^b). I found that this new curve, (a/x^b), fits our change in errors more precisely.

After implementing the cost-based optimizer with these two changes, I experimented with the different ways I can extend/modify this optimizer. I tried 4 different extensions to this optimizer.

1) Extension 1 (Attempt 1): The cost-based optimizer that these researchers designed is only choosing the GD algorithm that is most efficient in terms of it's *rate of convergence*. In other words, the optimizer *does not* choose the GD algorithm that'll reach the lowest loss in a specific amount of time. Their iterations estimator outputs an estimated number of iterations for a desired *change in error*. Change in error is what we use to measure *convergence*. (For example, a change in error of 0.001 means that the error between iterations was at most 0.001. This shows that the model isn't learning anymore, therefore it has converged).

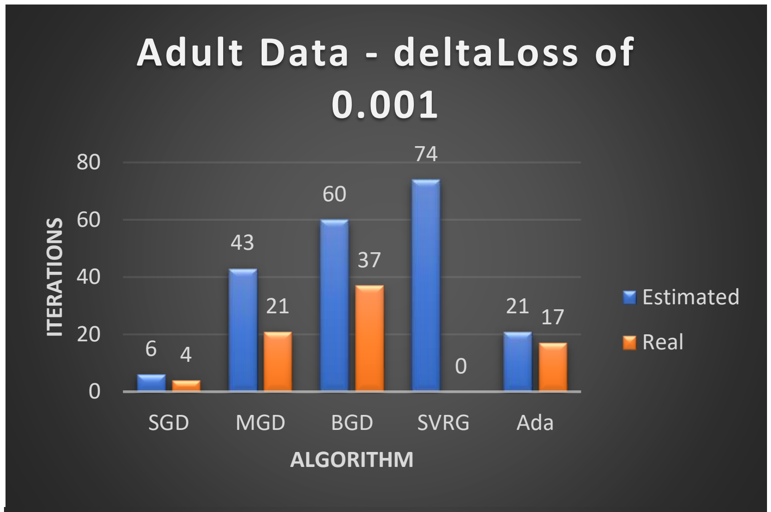
   I wanted to modify this optimizer, so it will choose the GD algorithm that is most efficient in terms of having a low error. All I had to do was keep track of the actual errors inside the iterations estimator, rather than keeping track of the change in errors. I was able to fit my list of errors to the curve (a/x^b), but I ran into a major problem when it came to testing the optimizer.

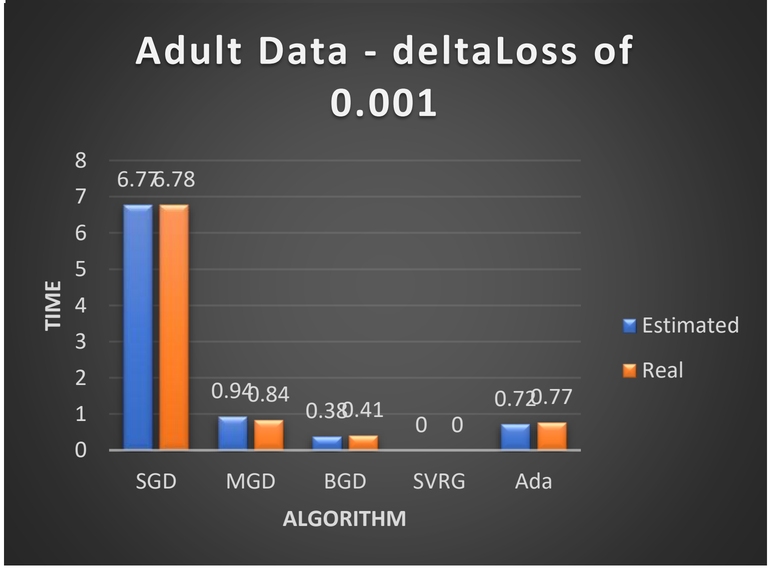
   The optimizer that the researchers came up with works well, because *all* GD algorithms will convergence. We can measure *when* a GD algorithm will convergence by looking at the change in error. My modification to this optimizer wasn't very successful, because we are not able to *always* measure the number of iterations when a GD algorithm will reach a specific error. This is because some GD algorithms are not able to reach a specific error due to converging early, so we won't be able to estimate the number of iterations.

2) Extension 2 (Attempt 2): After realizing that we are only able to have our optimizer choose the most efficient GD algorithm in terms of its *rate of convergence*, I looked into applying this for parallel implementations of GD algorithms, such as HogWild! I basically wanted to see if I can successfully estimate the number of iterations it takes for HogWild! to converge. This extension quickly failed due to me implementing this optimizer in Python. Python doesn't really allow true parallelism due to GIL (The Global Interpreter Lock). Because of this, I couldn't properly implement HogWild!

3) Extension 3 (Attempt 3): I decided I wanted to apply this optimizer to more modern GD algorithms. Instead of letting the optimizer choose between the three primitive types of gradient descent (BGD, MGD, and SGD), I wanted to let it to also consider stochastic variance reduced gradient descent (SVRG), and AdaGrad. However, I ran into a few

issues when getting some results. Again, since I implemented this in Python, some GD algorithms were performing *very* inefficiently. For example, SGD would take a while to run because it requires less use of a linear algebra operations library (like NumPy). I had to calculate the gradient for *each* example in SGD, which means I can't perform a matrix-vector operation, which is what we do in BGD. Because of this, there is more work done on the Python side, which causes a huge slowdown in performance. Therefore, SGD and SVRG have extremely slow runtimes. Below are some figures showing the optimizer with these additional GD algorithms implemented in Python. (Note: AdaGrad didn't have an extremely slow runtime because I set the batch size to 50.) (Note 2: At the time, my SVRG implementation was also a bit different from the one we implemented in Lab 3)





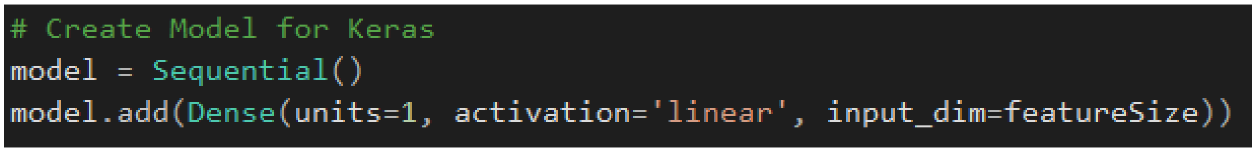
4) Extension 4 (Attempt 4, Success!): After seeing the slowdown caused by Python, and possible inaccuracy due to my own implementations of these GD algorithms, I decided to use a library that will perform the GD algorithm for me in a more efficient manner. Brian Tsan showed me a library (for Python) called Keras, which is built on top of TensorFlow. Keras allows us to run specific GD algorithms without the hassle of the setting things up

in TensorFlow. I went ahead and built my own GD algorithm optimizer on top of Keras, and I plan on making a pull request on their GitHub to see if they will like this contribution. This library allowed me to test my optimizer on the following algorithms: SGD, Momentum, Nesterov-Momentum, AdaGrad, AdaDelta, Adam, Adamax, Nadam, and RMSprop.

Implementation (Some brief code details)

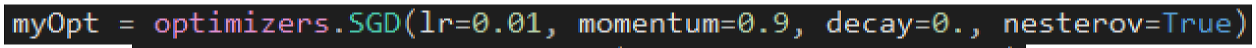
Implementing the iterations estimator with Keras was the bulk of the work. I used Keras in my iterations estimator to run GD algorithms on small subsets of the original dataset. I also used Keras on the full dataset to measure the average time per iteration. For this optimizer to be implemented in Python with Keras, I had to do the following:

1) Use Keras to create a model and run a specific GD algorithm: The first step is to create a Sequential model that has a Dense layer, which consists of 1 output and 300 input dimensions (w8a feature size).

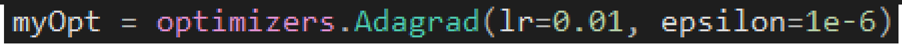


```
# Create Model for Keras
model = Sequential()
model.add(Dense(units=1, activation='linear', input_dim=featureSize))
```

Then we decide what kind of GD algorithm we would like to use. For example, let's say we wanted to use Nesterov-Momentum, or even AdaGrad.

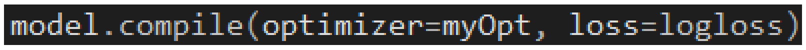


```
myOpt = optimizers.SGD(lr=0.01, momentum=0.9, decay=0., nesterov=True)
```



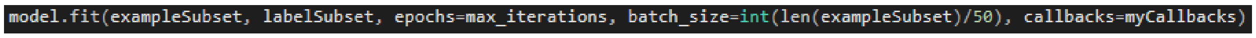
```
myOpt = optimizers.Adagrad(lr=0.01, epsilon=1e-6)
```

After we have set our desired GD algorithm, we need to compile our model.



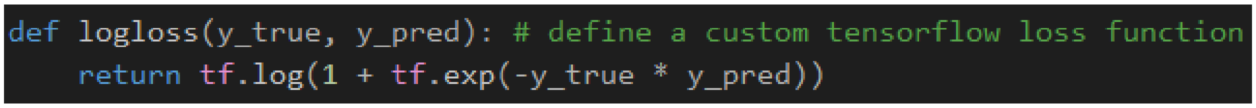
```
model.compile(optimizer=myOpt, loss=logloss)
```

Finally, we just need to train the model using the "fit" function.



```
model.fit(exampleSubset, labelSubset, epochs=max_iterations, batch_size=int(len(exampleSubset)/50), callbacks=myCallbacks)
```

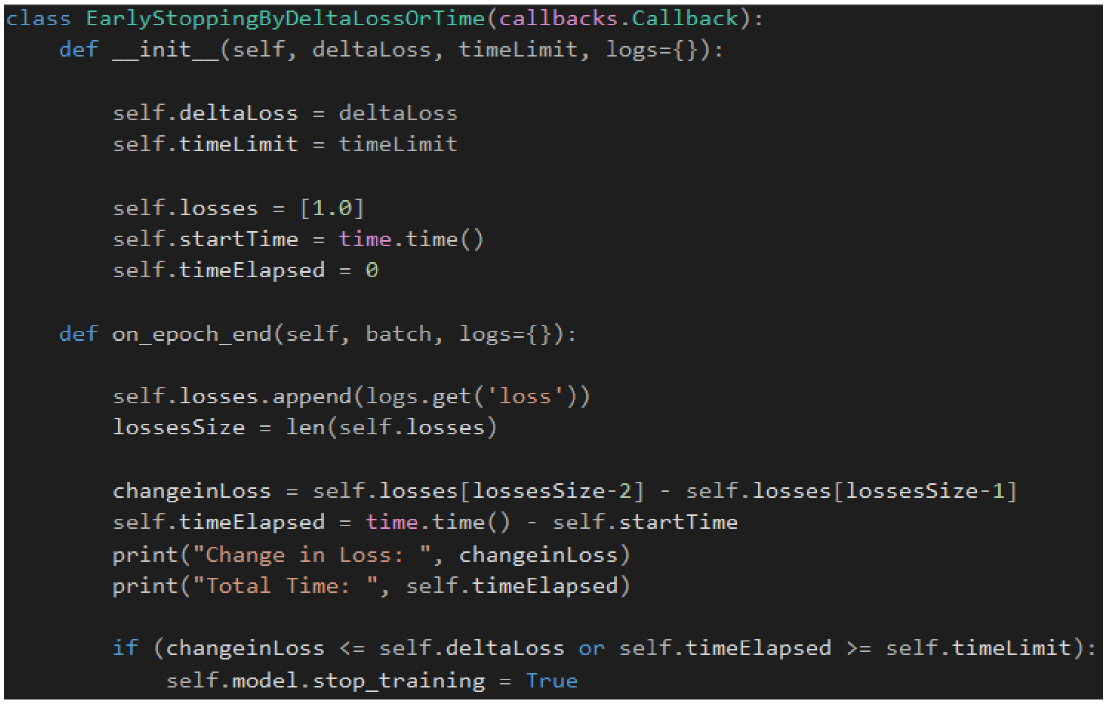
2) Use the TensorFlow language to create my own loss function for Logistic Regression: To have Keras run a GD algorithm on logistic regression, I had to define my own loss function using the TensorFlow API.



```
def logloss(y_true, y_pred): # define a custom tensorflow loss function
    return tf.log(1 + tf.exp(-y_true * y_pred))
```

(Credit: Brian Tsan)

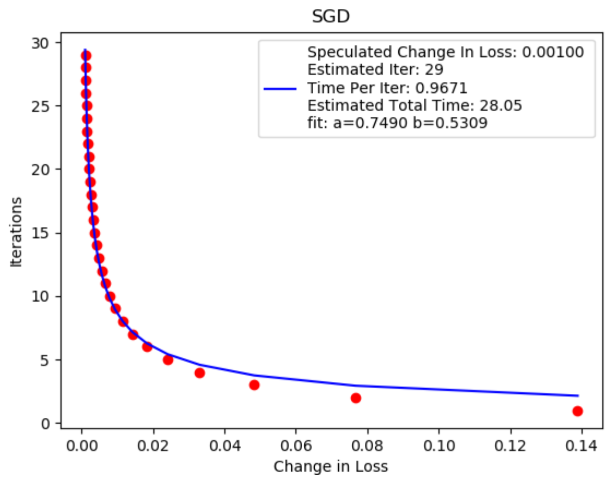
3) Write my own Callback function for Keras: The last thing I had to implement was the termination conditions. As stated earlier in the Related Work section, the iterations estimator has two termination conditions. The first condition is *reaching the desired change in error*, and the second condition is *reaching the time budget* allocated to this iterations estimator. When Keras trains the model, it only returns a History object. This History object contains all the losses for each iteration. I can use this to create my curve (a/x^b) that's needed to estimate iterations. However, this doesn't satisfy my termination conditions, because I only have access to this History object *after* the model is done training for the specified number of iterations. What if I need to terminate the training early due to the termination conditions? Because of this, I went ahead and wrote my own Callback function through Keras. A Callback function is called after *each* iteration of training. With my Callback function, I can check the change in error between iterations, and the total time elapsed.

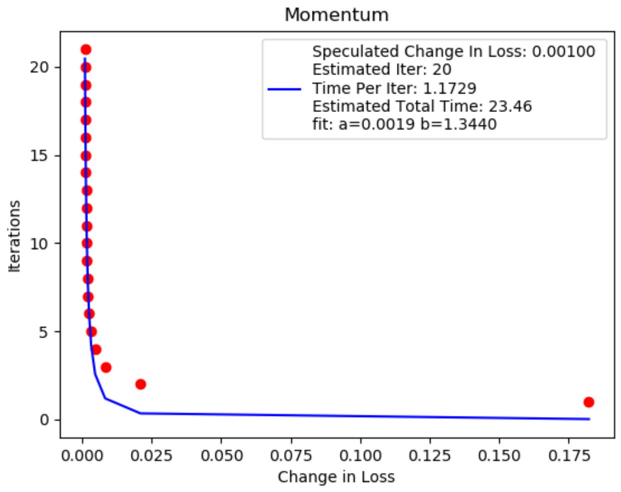


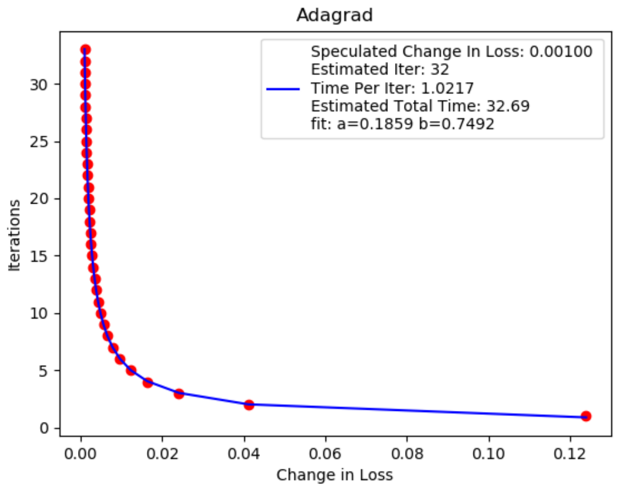
```
class EarlyStoppingByDeltaLossOrTime(callbacks.Callback):
    def __init__(self, deltaLoss, timeLimit, logs={}):

        self.deltaLoss = deltaLoss
        self.timeLimit = timeLimit

        self.losses = [1.0]
        self.startTime = time.time()
        self.timeElapsed = 0

    def on_epoch_end(self, batch, logs={}):

        self.losses.append(logs.get('loss'))
        lossesSize = len(self.losses)

        changeinLoss = self.losses[lossesSize-2] - self.losses[lossesSize-1]
        self.timeElapsed = time.time() - self.startTime
        print("Change in Loss: ", changeinLoss)
        print("Total Time: ", self.timeElapsed)

        if (changeinLoss <= self.deltaLoss or self.timeElapsed >= self.timeLimit):
            self.model.stop_training = True
```

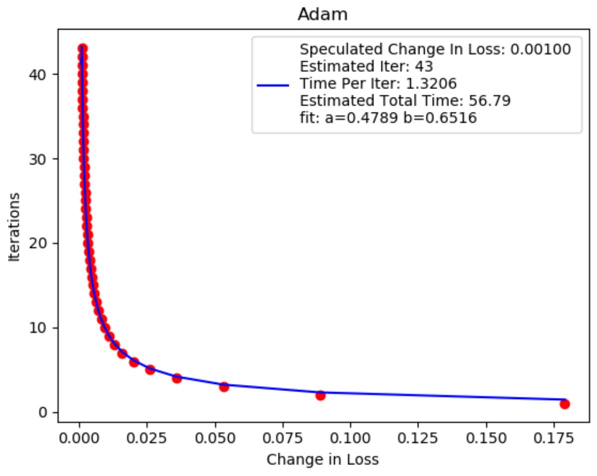
Experiments

1) Here are some figures showing change in errors fitted to the curve (a/x^b), for different GD algorithms:



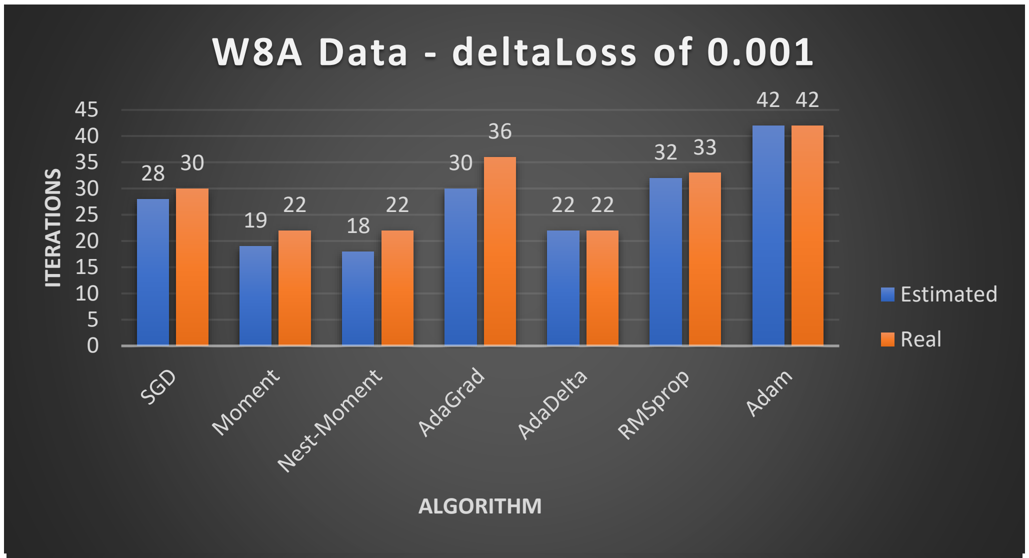


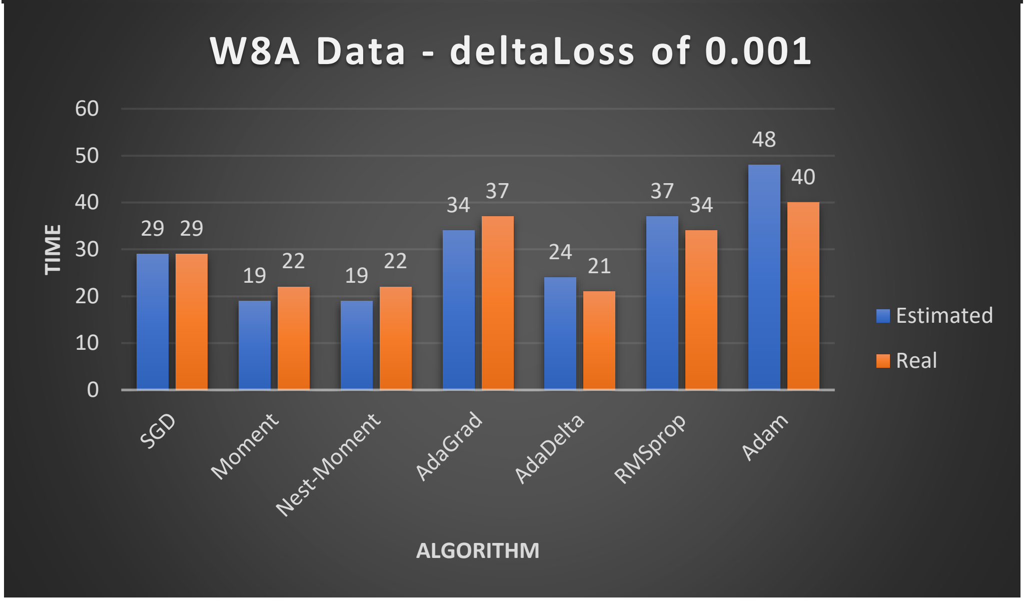




As shown in the figures above, we can successfully fit all types of GD algorithms to our curve, (a/x^b). This gives us the ability to accurately estimate the number of iterations required for convergence. The curve for Momentum looks like it doesn't fit the curve well. However, this curve is offset only at the *beginning* iterations of Momentum. As the model continues to train, the curve becomes more fit to the change in errors, which is what we care about (We want an accurate fit for small change in errors).

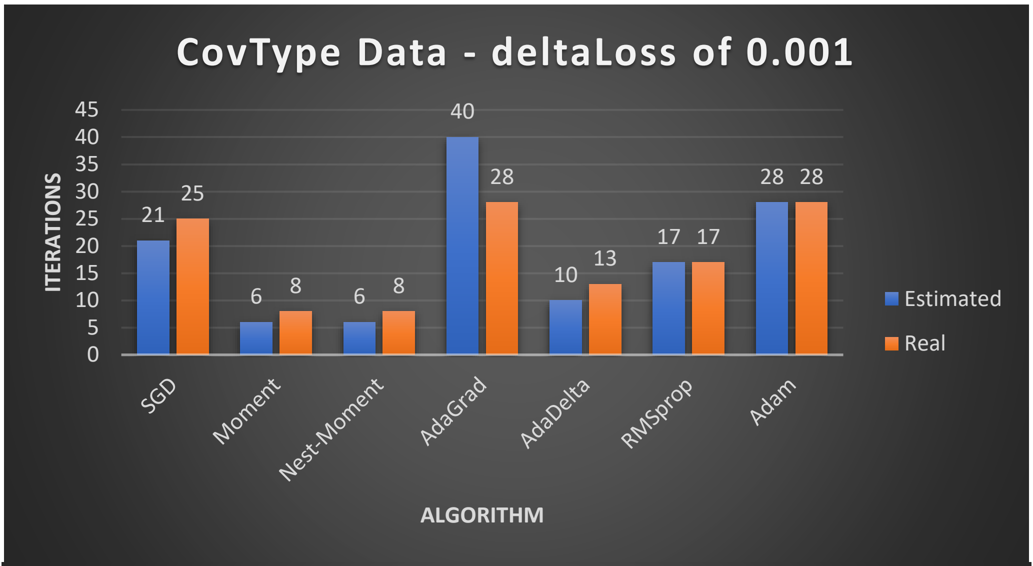
2) Iterations/Runtime Prediction Accuracy. W8A dataset: 59,245 examples 300 features

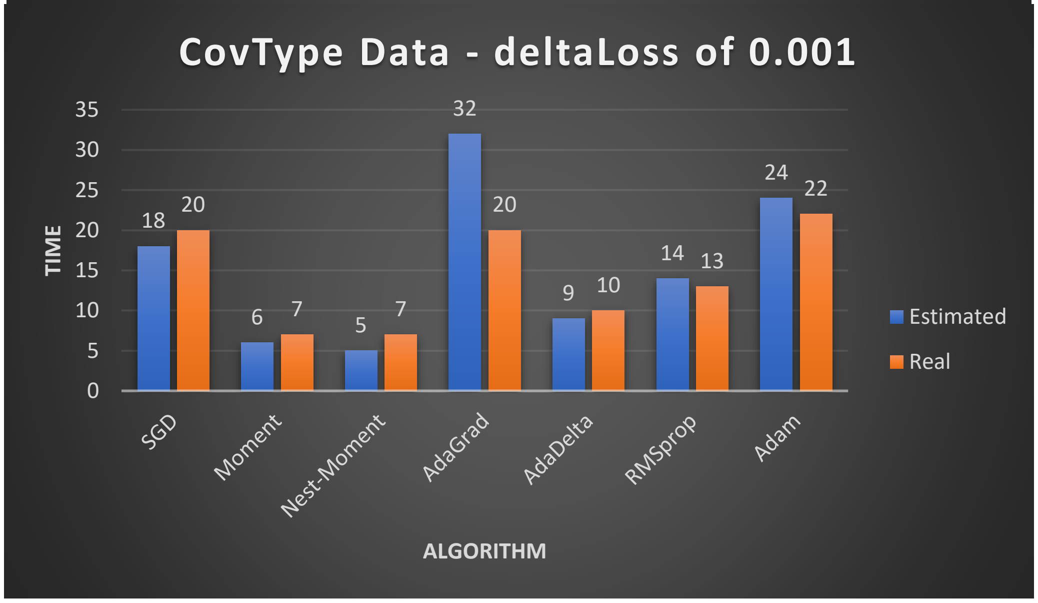




These two figures represent the estimated iterations/runtime and the actual iterations/runtime for a model to reach a change in error of 0.001 on the W8A dataset. For example, our optimizer predicted that SGD will require 28 iterations to reach a change in error (convergence) of 0.001. In actuality, it took 30 iterations. Keep in mind that the optimizer made this estimation based off a *small subset* of the original dataset. In other words, the optimizer only took around 20% of the original dataset and was able to make a very accurate prediction. The same follows for the other algorithms. The runtime for these algorithms were also estimated very accurately. The output of this optimizer would be Nesterov-Momentum, because that was the algorithm that reached a change in error of 0.001 the quickest. Similar figures are shown below.

3) Iterations/Runtime Prediction Accuracy. CovType dataset: 581,012 examples 54 features

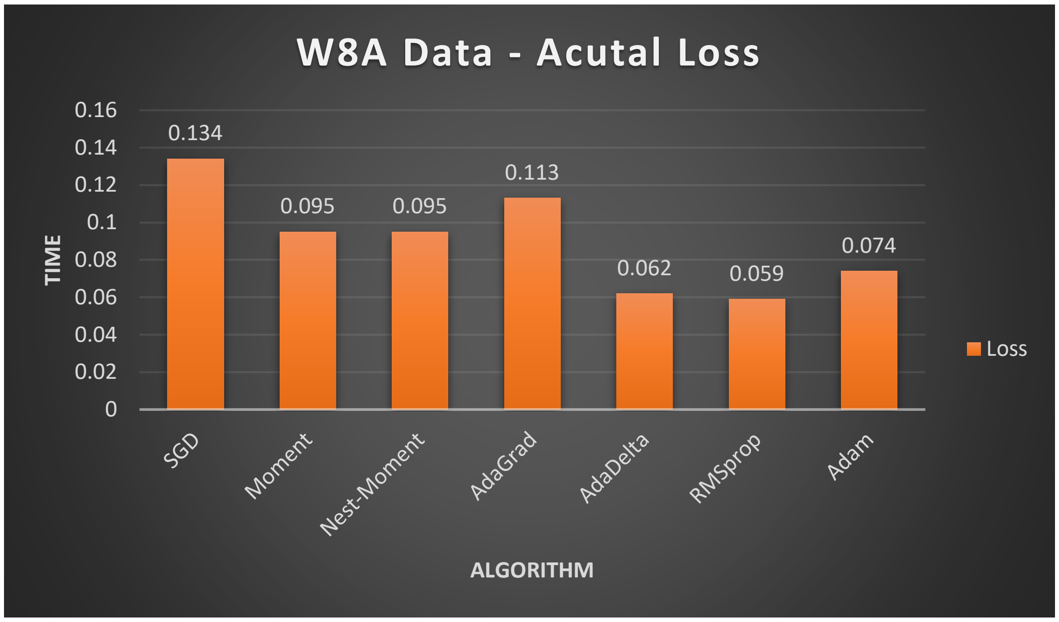




Findings and Conclusions

I was able to successfully build a GD algorithm optimizer on top of Keras in Python. It accurately estimates the number of iterations and runtime for each *modern* GD algorithm to reach a certain change in error. This helps automate the process of selecting which GD algorithm to use, since the optimizer will be able to choose the GD algorithm that converges the quickest.

The biggest problem with this optimizer is that it only optimizes the *convergence time* for gradient descent. It does not optimize the *minimum error* for gradient descent. For example, in the results that my optimizer produced, Nesterov-Momentum was considered the most efficient GD algorithm by my optimizer (See figures above). However, Nesterov-Momentum was only considered efficient for *quickest convergence*, but which GD algorithm was the most efficient in terms of achieving the *lowest error*? The figure below shows that other GD algorithms (such as AdaDelta, RMSprop, and Adam) can reach a better *minimum error* as compared to Nesterov-Momentum.



Here is some possible future work that can be done with this optimizer:

1) Test the optimizer on lock-free parallel GD algorithms like HogWild!
2) Train the optimizer to choose better hyperparameters. As of now, it just uses the default hyperparameters suggested by Keras.
3) Possibly extend the optimizer to find the most efficient GD algorithm in terms of *minimum error* rather than *quickest convergence*. Perhaps find the GD algorithm that can reach the lowest error with a certain time budget?
4) Test this optimizer on other types of models besides logistic regression. Perhaps SVM's?