Intelligently Evolving Stochastic Gradient Descent

Yousef Ahmad, John A. Vahedi

March 4, 2021

Abstract

We attempt to implement a more intelligent form of the gradient descent algorithm, which steps semi-stochastically as informed by the information collected during its history. These informatics are used as a type of fitness in a evolutionary method that determines how partial stochasticity is implemented. The main information used is the partial gradient sizes contributing to the the last stepping stage. Three methods are implemented, ranging from simple to more involved. The robust algorithms more closely follows an evolutionary algorithm that has a large population in memory and track fitness for each member of the population. The simpler method is not population-based but incorporates a type of gradient fitness that 'remembers' which dimensions contributed to the gradient more effectively. These methods are compared to the 'vanilla' SGD in both linear and nonlinear examples after effective parameter tuning. The ultimate goal of this method is to train parameters more effectively, for stable but complicated neural network and transformer architectures.

1 Introduction

Here we introduce two independent ideas to the reader that will be required in implementing our method. First we begin a review of the basic 'vanilla' gradient descent method used in many optimization problems. We briefly include a run through of how one can use stochastics to improve this algorithm. Next we will introduce you to how evolutionary algorithms operate, as inspired by biology, to find the fittest individuals in an environment.

1.1 Gradient Descent

The method behind gradient descent is a subclass of a bigger set of methods called Robbins-Monro methods. Yet our version has a visually compelling reason for working. Say I have a function f that I want to find its minimum. Let f depend on a set of parameters w. (I am more specifically interested in finding the parameters w that help me arrive at this minimum.) Imagine for each set of these parameters w you create a resultant value for f. Say I have only two parameters in w and for every such combination I calculate my f, and in this way I create a landscape. This is similar to imagining your parameters w are latitude and longitude and the value of you f would be your, in place, altitude. Similarly, for a parameter space of higher than two you would create a hyperplane that would result in place value of f for each such combinations.

1.1.1 Vanilla

Now let us begin with a specific set of these parameters w which would drop me in a specific spot on this terrain. If I changed each of these individual parameters by a little bit, this would result in a little change in my f which can be translated as incline in each of those specific directions. We can capture all this information in a gradient $\nabla_w f(w)$ which reads as the gradient of f over the directions w and which is effectively a vector of inclines over our parameters. If we look at this vector directly it is easy to see that it points in the direction that raises me most over each direction. Since any well behaved surface is locally planer we can assume that this direction is most advantageous in increasing my function value, at least locally or at least before the curvature effects come into play further out. The reverse is similarly true, the opposite direction to my vector, the negative gradient, will direct me in the direction that decreases my function value most, again locally. This is the idea behind gradient descent, we find our local gradient, step in that direction negatively and then reevaluate once i have stepped into my new location. The analogy can be made to someone trying to get to the bottom of a small mountain but finds herself in a thick fog! So she has only a local sense of what direction will take her down the most and keeps reevaluating as she steps further down and hopes this will get her down eventually to the lowest point.

We can capture this process as:

$$w_{i+1} = w_i - \eta \cdot \nabla_w f(w_i) \tag{1}$$

where w_0 is my initial choice of parameters, w_i is my value for my parameters on my ith update, and eta is the stepping size also sometimes referred to as the 'learning rate' by those nasty data scientists. A termination conditions can be either a hard number of iterations $i = N$ or when the size of the gradient is vanishingly small and therefore it makes sense to quit the algorithm as little learning is happening at that point. Yet this may indicate you have reached a local minimum or that you have reached a small plateau which has unfortunately and dangerously allowed for learning to stop even though the minimum may be next door. Since you have linked learning with the gradient directly you have opened yourself up to this inadvertent risk. You can fix this by introducing a noise term to the overall expression. This help roll your position out of its rut and hopefully towards a steeper incline that will continue your algorithmic descent. We can model this with an added noise term ϵ, that can be a small multidimensional Gaussian:

$$w_{i+1} = w_i - \eta \cdot \nabla_w f(w_i) + \epsilon \tag{2}$$

This may help avoid this risk but may also slow down our time to convergence due to the random walk I make at all times. Yet this can be implemented more intelligently by give it some dependence as well which we will discuss later on.

1.1.2 Step Size η

Before moving on, we must discuss one of the critical ways any gradient descent technique can be improved and that is regarding how η the step size is chosen and implemented. First of all, η may not be too big as it will lead to divergent behavior. The reason for this is that often the 'landscape' is such that if one of your parameters blows up in size so does your function value. As we will discuss later, one such example of our function f which has this behavior is the cost function Φ. As we get away from our minimum position so does our function. The first *faux pas*: what this means is that if you overstep and over shoot your minimum location, you may end up at a far out position that has a higher gradient, which in turn sends you further out and so on, eventually towards divergence. Now if we go in the other direction we could take small step sizes to avoid over shooting our target and possibly sending myself out of bounds. The second *faux pas*: yet if you take underwhelming step sizes it my take you an indeterminate time to reach your position and can be computationally expensive. In other words you want to take step sizes that are large enough to make fast progress but without falling into the trap of blowing up.

Therefore one would imagine that some intermediate step size which one could experimentally determine would work nicely. The third *faux pas*: while you may get lucky and get close enough using this framework, in general you are likely to bounce around the minimum towards the end as your step sizes never get smaller or small enough to increase accuracy at some stage. Therefore a better idea is to somehow effectively decrease our step size towards the end or later on in the process so that we are better able to sink into the minimum if the gradient does not do this automatically. Therefore we would want η_i to dependent on the iteration count i. An effective and simple way to do this is to let:

$$w_{i+1} = w_i - \eta_i \cdot \nabla_w f(w_i) + \epsilon, \qquad \eta_i = \frac{\eta_0}{i^p}, \qquad p \in [0, 1]$$

Here η_0 is a free parameter that can be tuned like before, and where p is the strength of how fast it will drive the step sizes down, higher values of p correspond to faster decreases. We should note that one is forced to choose $p \in [0, 1]$. One can easily see why we do not allow $p < 0$ as it would cause our stepping size to grow. On the other hand allowing $p > 1$ risks a trivial convergence as the sum of such a series is finite and is considered a bad idea.

Additionally one can prove that you are guaranteed convergence to some minimum of a "well-behaved" function if $p = 0.5$.

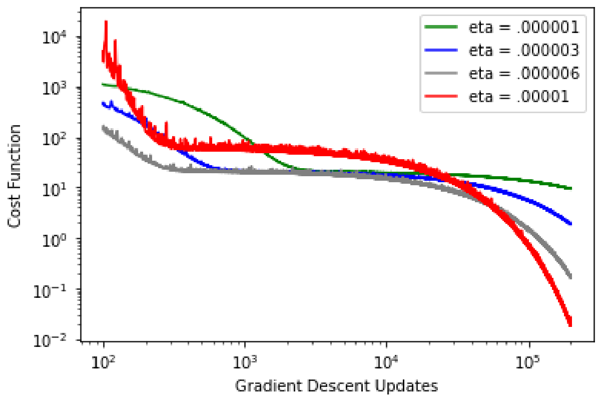

Figure 1: η's effect on convergence - interestingly, convergence rates vary across epochs of iterations. May be due to local minimum.

1.1.3 Stochastic

After establishing the concepts behind a standard gradient descent, we can now understand how stochasticity can be introduced. The basic idea behind this method is to search for a subset of the true gradient – at random. What we mean by at random we will delineate a bit later. One might ask why would I do this, specifically: why would I choose only to step only over a subset of the gradient instead of the whole, and would this not take me in a roundabout or unreliable direction? An obvious response to the first question is that it is often far too expensive to calculate the entire gradient each time instead of over a cheaper small subset. While this gives motivation in regards to first question it does not explain why this a good idea beyond a computational argument.

To motivate this let us return to what we meant by "random subset." One way to understand the subset is over the parameters w, meaning we only calculate gradients from a small subset of these parameters in our step. The reason why this may be advantageous is that globally some of our parameters do not play a strong role in predicting our objective and we are taking advantage of this by skipping out some of these from time to time. Alternatively there may be a local independence from some parameters contributing. Worse case scenario is that we may have to run more iterations to recreate the complete information – but we may get lucky and that is why we attempt this. In general or in an applied problem such as model fitting, when models are set up, they are not done intelligently and therefore many of the parameters will not contribute in an important way. We are banking on this on average to hopefully speed up our calculation. A less extreme understanding of this is that our parameters are not contributing orthogonally or independently.

This is not the only way to understand what we could mean by subset. Let us first explicitly construct our function f, the cost function we will introduce as Φ. This cost function is constructed from many smaller cost functions ϕ_j. In a regression type problem there would one ϕ_j for each each of your M data points (x_j, y_j) for $j \in \{1, ..., M\}$. Here we choose the squared loss function as our cost function (which we intent to minimize). First we will need a model to fit the data by parameterization. A **linear** model could look like (including the gradient):

$$F(a,x) = a_1 \cdot f_1(x) + a_2 \cdot f_2(x) + ...a_N \cdot f_N(x) = \sum_{n=1}^{N} a_n \cdot f_n(x) \tag{3}$$

$$\nabla_a F(a,x) = [f_1(x), f_2(x), ..., f_N(x)]^T \tag{4}$$

as it is linear in the parameters a and $f_n(x)$ can be any function of choice. Next our smaller cost functions squared loss will be:

$$\phi_j(a) = (y_j - F(a, x_j))^2. \tag{5}$$

$$\nabla_a \phi_j(a) = 2 \cdot (y_j - F(a, x_j)) \cdot \nabla_a F(a, x_j) \tag{6}$$

Our total loss function (or the cost function) which is better interpreted as an average of these smaller cost functions is:

$$\Phi(a) = \frac{1}{M} \sum_{j=1}^{M} \phi_j(a). \tag{7}$$

$$\nabla_a \Phi(a) = \frac{1}{M} \sum_{j=1}^{M} \nabla_a \phi_j(a) = \frac{1}{M} \sum_{j=1}^{M} 2 \cdot (y_j - F(a, x_j)) \cdot \nabla_a F(a, x_j) \tag{8}$$

In this framework, we apply stochasticity differently. Instead of choosing which parameters a contribute to the gradient, what we choose randomly is the random subset of the data that will contribute to the gradient. In other words before our w was just the parameters a that we randomly chose from. Now we can imagine each smaller cost function having a parameter w_j in front that can be turned on or off: $w_j \in \{0, 1\}$. Therefore we have:

$$\Phi^*(a) = \frac{1}{\sum_{j=1}^{M} w_j} \sum_{j=1}^{M} w_j \phi_j(a) = \frac{1}{K} \sum_{j=1}^{M} w_j \phi_j(a) \tag{9}$$

where the number of data point in the random subset is K. What we intend for this method is that

$$\nabla_a \Phi(a) \approx \nabla_a \Phi^*(a).$$

$$\nabla_a \Phi(a) = \nabla_a \Phi^*(a) + err, \qquad err \sim N(0, \sigma) \Rightarrow$$

$$E[\nabla_a \Phi(a)] = E[\nabla_a \Phi^*(a)]$$

In data often, not each data point is uniquely robust and therefore our data is largely redundant. In other words, if we choose a random subset of the data, or the $\phi_j's$, we are likely to get a vector pointing in a similar direction to the true gradient with some noise, as it is off by some bit. What we mean is that data no matter what subset you choose will largely point you in a similar direction on how to update your parameters a. This way we avoid an expensive calculation by finding the contributions arsing from what may be a very large number of data points. As an analogy, you can imagine if you need some advice you can ask both your father and mother. Yet there is a good chance the may give you similar advice. So you may choose just one to ask and if only necessary then to follow up with the other. Similarly we choose only a subset of the data to get "advice" from in the form of a gradient on how to update my parameters a. Note that in this structure all a parameters are included and no random subset is applied to them (although in theory you could do this to both simultaneously). Our ultimate update scheme is:

$$a_{i+1} = a_i - \eta_i \cdot \nabla_a \Phi^*(a_i) \tag{10}$$

1.2 Evolutionary Algorithms

The next technique that we incorporate is the evolutionary algorithm. Evolutionary algorithms operate on optimization problems by employing processes that imitate how organisms evolve. These algorithms are population-based, which means that they consider multiple solutions to the problem at hand in parallel. We define a set of these candidate solutions as a "population". The evolutionary aspect of these algorithms is in the fact that, after a population is evaluated, the top-performers are selected and are made to "breed". This process repeats itself, yielding a cycle which is analogous to the survival of the fittest. Below, we provide some more detail on breeding. Information on evolutionary algorithms, beyond what we discuss in this paper, can be found in [2].

1.2.1 Breeding Mechanisms

There are two core types of breeding mechanisms that are most common in evolutionary algorithms. These mechanisms are mutation and crossover. Mutation entails making very minor, random modifications to a member of the population, similar to hill-climbing search methods. Crossover is a little more involved - this mechanism essentially involves combining one member of the population with another member. More specific analysis on these two mechanisms can be found in [1].

2 Evolutionary SGD

In the following section we introduce a novel method to improve the SGD algorithm. Simply at this point we have taken advantage of stochasticity to simplify the complexity of gradient vector calculation in an approximate way. Yet each step we take a complete new random subset. What if instead we used our previous history to make smarter choices about how to choose these subsets? Our history can be stored in memory and used to make decisions further down the line. This will result in what we call a semi-stochastic gradient descent as will still allow for a guided stochastic descent given some historical context. The following three methods build on this idea from simpler to more complicated forms and from less guided to more guided.

2.1 Simple Method

The simplest method calculates the size of the sub-gradients ϕ_{j_k} arising from each of the data-points in the subset and stores in these intermittently to act as a short term type of 'fitness.' The hope is that knowing the relative contributions of each data point at this step will allow us better to predict what may continue to be important local contributors. Once we have the sizes of the sub-gradients we sort and keep the top T contributors and re-randomize the rest of the subset. In this way we re-introduce the idea of the subset and stochasticity but one that 'remembers' which few performed best and stays with them which coins the algorithm as a "Semi-Stochastic Gradient Descent". In our code we made this 'remembering' about the data point as was our general stochasticity. You could as well implement this over the parameters a.

Algorithm 1 SSGD

```
1: Import data
2: Initialize subset of data, size $K$ - call this subset $L_k$
3: Initialize $a$ at initial guess $a_0$
4: for $i = 1, 2, \dots, N$ do
5:     Set $\nabla_a \Phi$ = zeros(input space dimensionality)
6:     for $point = 1, 2, \dots, K$ do
7:         Assess gradient at point $L$[point] and add to $\nabla_a \Phi$
8:     end for
9:     Update a=a$-(\eta/i^p)$*($\nabla_a \Phi/K$)
10:    Selection on $L_k$, return $L_{k,selected}$ holding highest-contributing points
11:    Repopulate $L_{k,selected}$ with random points from dataset, set repopulated
       array equal to $L_k$
12: end for
```

2.2 Methods for More Complex Datasets

In our more complex methods, we incorporate evolutionary algorithms as described in (Section 1.4). Specifically, we add onto the "simple method" in (Section 2.1) by maintaining a population of solutions and having this population breed after selection. The population at hand pertains to different subsets of data - with each member of the population being a subset, we can think of this as being the same as (L_k as described above)

We found one method which combines stochastic gradient descent and evolutionary algorithms [4] - this was interesting. However, this method ultimately does not incorporate evolutionary algorithms in the way that we do - they intersperse stochastic gradient descent steps with evolutionary algorithm iterations in a back-and-forth process in which both gradient descent and stochastic gradient descent work towards optimizing the same parameters. We, on the other hand, perform selection and evolution on selecting the batches of data on which we perform gradient descent. Thus, in our scenario, evolutionary algorithms are merely used to inform the batches that the gradient descent operates on.

The motivation behind making batch selection a process driven by evolutionary algorithms, rather than a purely stochastic one, is in the idea that, at different points in parameter space, there should lie some logic to the question of which input data to use. Explicitly coding out this logic would be expensive, and using an evolutionary algorithm to arrive at that logic allows the software to come to this logic itself. In order to find these data points that contribute most to our larger optimization problem of our parameter space, we felt appropriate fitness criteria for a member of the population would incorporate: (1) the member's accumulated gradient value ($\nabla_a \Phi$), (2) the absolute error between the function produced by the larger optimization problem and member data points, and (3) the age of the member. High accumulated gradients are incentivized - involving accumulated gradients is necessary as it helps decide which batches of data drive our larger optimization farthest. High error is penalized - including error ensures that we do not look for batches which push our larger optimization arbitrarily based on magnitude (effectively, this provides balance to the prioritization of large gradients). Finally, older solutions (i.e. subsets) are penalized - this is included to temper the error criteria, as we feel that incorporating age avoids the risk of local minima. These factors come together in the below fitness function:

$$\text{fitness} = \nabla_a \Phi * \frac{1}{max(\text{age}, 5)} * \frac{1}{\text{error}} \tag{11}$$

An element of stochasticity is also retained, in that, when breeding, the mutation operator selects random elements from the input dataset.

We decided on two variants on how to use this evolutionary selection of batches. The first of these variants (Comprehensive EvSGD: CEvSGD) tracks the larger optimization's results for each member of the population, essentially holding the larger optimization's parameters for the current population in memory. This means that there is a direct mapping from larger optimization results to population members (i.e. subsets of data). The second variant (Efficient EvSGD: EEvSGD) does not hold this mapping - instead, it averages out the gradients that each member (subset of data) in the population generates and uses this to perform the gradient descent update for the entire population. To summarize, CEvSGD holds the gradients for each member of the population while EEvSGD averages them out. Practical outcomes of this are a decrease in computational load for EEvSGD, but more robustness for CEvSGD.

Algorithm 2 EvSGD Variant 1: CEvSGD (Comprehensive EvSGD)

```
1: Import data
2: Initialize $P$ subsets of data, each of size $K$ - call this array $L$
3: Initialize array $A$ of size $P$, entries equal to initial guess $a_0$
4: for $i = 1, 2, \ldots, N$ do
5:     for $member = 1, 2, \ldots, P$ do
6:         Set $L_k = L$[member]
7:         Set $\nabla_a \Phi = zeros$(input space dimensionality)
8:         for $point = 1, 2, \ldots, K$ do
9:             Assess gradient at point $L_k$[point]and add to $\nabla_a \Phi$[point]
10:        end for
11:        $a = A$[member]
12:        Update $a$=$a$−$(\eta/i^p)$*($\nabla_a \Phi$[member]$/K$) and correspondingly update $A$
13:    end for
14:    Selection on $L$, return $L_{selected}$ which holds top performing subsets
15:    Run breeding on $L_{selected}$, return $L_{new}$
16:    Update $A$, with new members of the population initialized at $A$[parent]
17: end for
```

Algorithm 3 EvSGD Variant 2: EEvSGD (Efficient EvSGD)

```
1: Import data
2: Initialize an array $L$ of $P$ subsets of data, each of size $K$ - $L$ corresponds to
   our population
3: Initialize $a$ equal to initial guess $a_0$
4: Initialize an array $\nabla_a \Phi$ of size P, each entry is $zeros(P)$
5: for $i = 1, 2, \ldots, N$ do
6:     for $member = 1, 2, \ldots, P$ do
7:         Set $L_k$ = $L$[member]
8:         Set $\nabla_a \Phi$[member] = $zeros(P)$
9:         for $point = 1, 2, \ldots, K$ do
10:            Assess gradient at $L_k$[point] and add to $\nabla_a \Phi$[member]
11:        end for
12:    end for
13:    Selection on $L$, return $L_{selected}$, which holds top-performing subsets
14:    $\nabla_{a,avg}\Phi = average(\nabla_a \Phi)$
15:    Update $a = a - (\eta / i^p) * (\nabla_{a,avg}\Phi / K)$
16:    Run breeding on $L_{selected}$, return $L_{new}$
17: end for
```

3 Results & Discussion

We would like to point out that all these results were obtained without importing packages beyond numpy as we maintained fine control by writing all the code from scratch. Link at the end of the paper.

3.1 Linear Results

3.1.1 Cost Function

Our cost function is shown again below.

$$\Phi(a) = \frac{1}{M} \sum_{j=1}^{M} \phi_j(a). \tag{12}$$

For each of our methods, we use this cost function as a direct metric for how well a solution performs. Below, we include figures that show our performance, as well as that of a vanilla SGD benchmark, by means of our cost function. The evolutionary algorithms had population sizes of 4, and every method (vanilla included) used subsets of 25 percent of the original dataset for gradient descent updates. It is very important to note that, while CEvSGD outperformed all other models, it was also found to be the most unstable in that it often diverged (discussed further in conclusion).

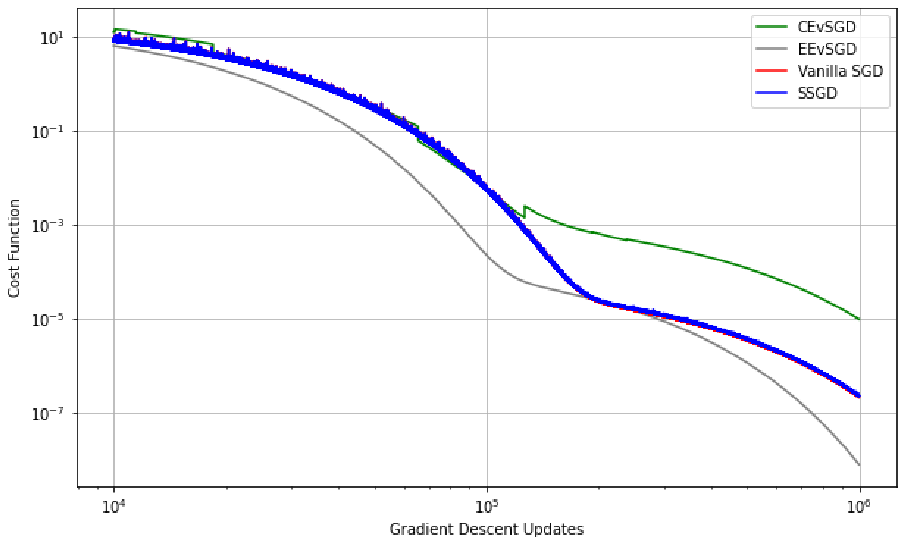

Figure 2: Cost Function, 1m Iterations (5 runs averaged)

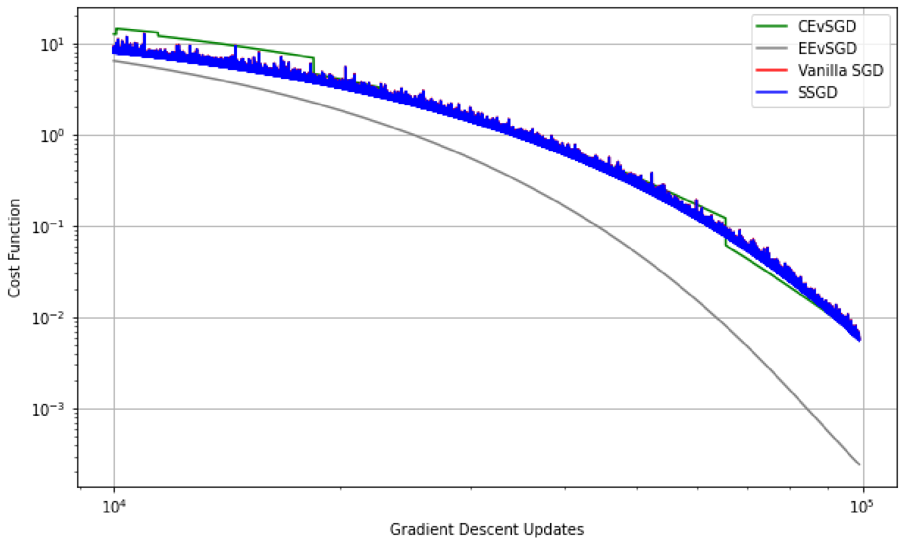

Figure 3: Cost Function, 100k Iterations (5 runs averaged)

3.1.2 Regression

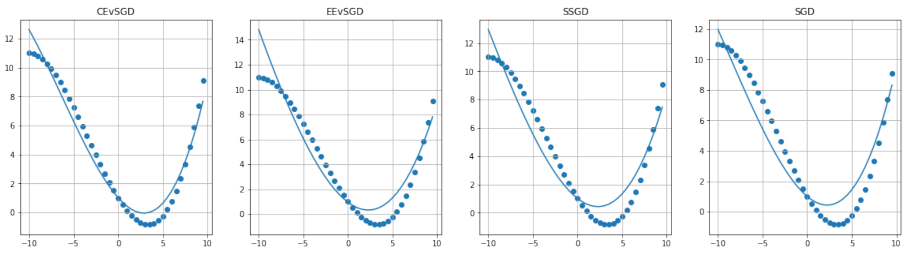

Figure 4: Models regressed over dataset at 30k steps (above)

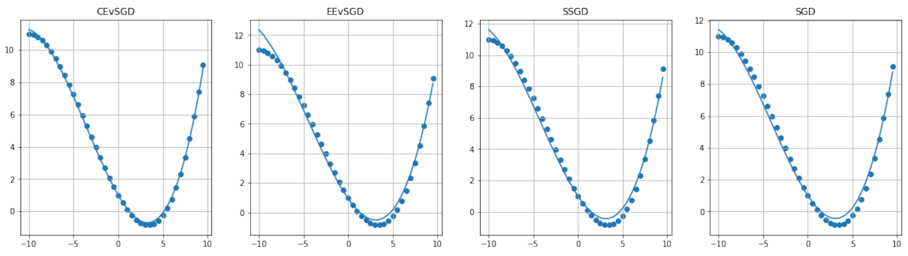

Figure 5: Models regressed over dataset at 60k steps (above)

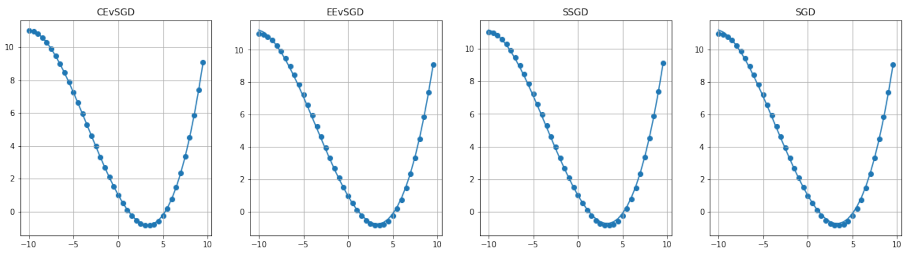

Figure 6: Models regressed over dataset at 90k steps (above)

3.2 Time Taken

As can be seen below, time efficiency was the area in which our models do not perform very well.

Algorithm	Time Taken (Percentage of Time Taken for SSGD)
1: SSGD	100
2: EEvSGD	125
3: CEvSGD	126
4: Vanilla SGD	42

We have two main hypotheses behind why this is the case: (1) our models are slow due to inefficient code, and (2) our models are better for more complex datasets. Inefficient code comes from the fact that neither of us are software engineers, and so all of our mechanisms (such as sorting, for example) can likely be made more efficient. Since vanilla SGD does not need any advanced mechanisms, our inefficient code does not bog it down in the way that it bogs down the novel methods. We discuss the complexity aspect in (Section 4.1.1) - in a nutshell, one would still expect vanilla SGD to run more quickly, but the question is whether it outperforms our methods on an architecture like that of a transformer.

For example in SSGD we use an L_2-norm to get the size of the gradient contributions from a point. We could calculate an L_1-norm for efficiency with somewhat similar results.

3.3 Tuning and Stability

3.3.1 Parameter Tuning

When we began testing out our methods, we were surprised by just how sensitive models were to hyperparameter choice. We found that results would often tend towards infinity. Further digging showed us that the cost function reflected this instability - depending on the step taken by the gradient descent algorithm, the cost function could get out of hand. Repeated missteps would yield results spiralling out of control. The following figure shows an example of a time where the gradient descent updates jumped around a lot.

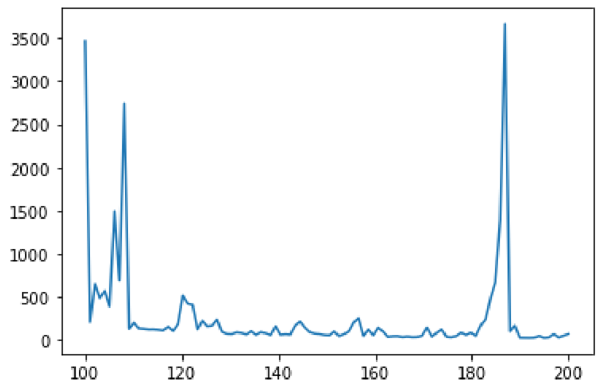

Figure 7: Early Cost Function Shows Jumpy Behaviour - Iterations 100 through 200

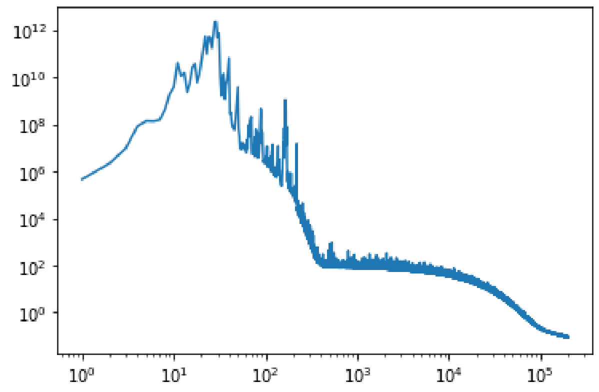

Figure 8: ... and it gets worse before it gets better

To minimize the effects of this unpredictability (born from stochasticity), a significant portion of our time was devoted to understanding the hyperparameters to be passed to our models. As can be seen below, our choice in p and η - the stepping parameters outlined in (Section 1.2.1) - could make or break the model's results.

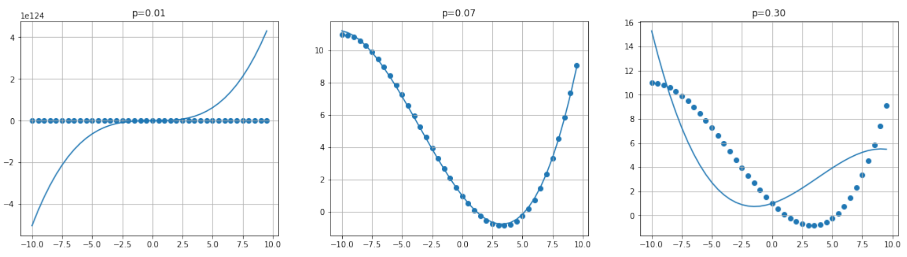

Figure 9: Effect of different p values for a fixed η=0.000001 on SSGD

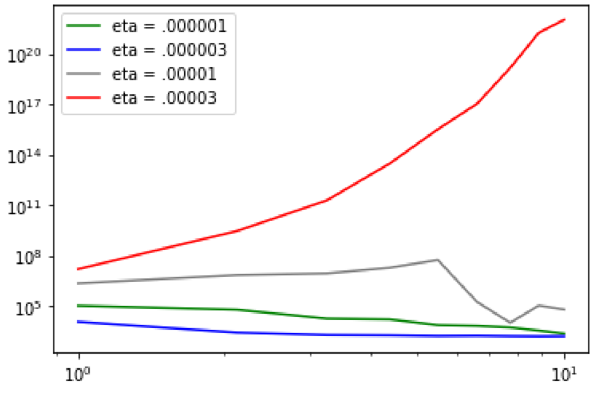

Figure 10: Effect of different η values on cost function for a fixed p=0.1 on SSGD

Clearly, a sweet spot must be hit for any given η or p - in the previous example, this sweet spot was at a p value of 0.07.

In order to tune our parameters to find an "optimal" combination of parameters, we ran grid searches. As previously mentioned, the key parameters of interest here were η_0 and p. This was an area which we realized was crucial to improving optimization in the way that we hope to (which is to speed up convergence). After running a grid search on $p \in [0 : 0.5]$ and $\eta_0 \in [1 \times 10^{-7} : 5 \times 10^{-5}]$, we found the following combinations of p and η_0 worked best.

Algorithm	Optimal p	Optimal η_0
1: SSGD	0.07	$9{\times}10^{-6}$
2: EEvSGD	0	$1.5{\times}10^{-7}$
3: CEvSGD	0.07	$8.1{\times}10^{-6}$

For users of the algorithm, we recommend first choosing a value for p, as this can serve as a measure of how aggressive the user wishes to be with the algorithm. Low p corresponds to being safer - high p generally means starting with a larger η_0 is fine as the step size will decrease after some early, large steps.

After choosing a p value, one can then tune η_0 through a search algorithm (on a space similar to the one we used on our grid search).

3.3.2 Standardization

One common technique in SGD is to "normalize" your features in an attempt to not privilege one of your dimensions to create a high gradient. We attempted a mean and standard deviation transformation on each of our features. We assumed this would result in the learning of special a^* parameters that would map the (once) normalized inputs in the future to the outputs correctly. We were interested in finding the original parameters of the model, once we figured out how to transform back after. Once again data scientist do not care for this. So we delineate the process below in a cubic model as an example.

Normalized Discovered Version:

$$
\begin{aligned}
y &= a_0^* + a_1^*(X)_s + a_2^*(X^2)_s + a_3^*(X^3)_s \\
&= a_0^* + a_1^*\frac{(X-\mu_1)}{\sigma_1} + a_2^*\frac{(X^2-\mu_2)}{\sigma_2} + a_3^*\frac{(X^3-\mu_3)}{\sigma_3} \\
&= (a_0^* - \sum_{i=0}^{3}\frac{a_i^*}{\sigma_i}) + \frac{a_1^*}{\sigma_1}X + \frac{a_2^*}{\sigma_2}X^2 + \frac{a_3^*}{\sigma_3}X^3
\end{aligned}
$$

Note that we chose not to apply the standardization to the first feature for obvious reason. Therefore we have the transformation back as:

$$
a_0 = a_0^* - \sum_{i=0}^{3}\frac{a_i^*}{\sigma_i}, \qquad a_i = \frac{a_i^*}{\sigma_i}
$$

Unfortunately this transformation did not get us back to our correct values and we did not clarify as of this point why.

3.4 Non-Linear

We included nonlinear functions in the code but decided that there were many other complications to the implementation and that the linear problem was rich enough as is. Yet the non-linear problem may have different advantages and disadvantages.

4 Conclusion

4.1 Future Methods to Explore

4.1.1 More Complex Models

The problems we tested our methods on were ultimately quite simple. The end goal, however, as stated previously, would be to use this optimization method on a more complicated problem. Such problems include architectures for large models, such as neural networks and transformers. We anticipate that EEvSGD and CEvSGD will adapt well to such architectures, due to the more informed manner in which batch selection is handled (i.e. evolutionary algorithms' stochastic component being less prominent than SSGD's). We believe that the extra computational strain imposed by EEvSGD and CEvSGD should be offset by quicker convergence. However, from the models we developed, SSGD offers better computational efficiency and is more appropriate for simpler models.

Additionally, it could be interesting to test these methods on more lopsided data. Again, due to the more informed batch selection, we believe EEvSGD and CEvSGD would scale well to this scenario. This is quite a simple test, and is one we would certainly have run had we more time on our hands.

4.1.2 Modifications on Evolutionary Elements

Clearly, the fitness function we use for our evolutionary algorithm is quite simple. This is the the most logical item to tackle as far as improvements to the EvSGD algorithms go - combining age, gradients, and error in a more intelligent manner could certainly improve performance.

Beyond this, there are many alternative, more complex evolutionary algorithm strategies to be attempted.

4.1.3 Stability and Unpredictability

Possibly the most interesting thing regarding our results was the vastly superior performance of our CEvSGD algorithm. It is important to note that this was also the most unstable algorithm - it yielded unstable (i.e. tending toward infinity) solutions roughly fifty percent of the time. On the other end of the spectrum, SSGD and EEvSGD had the same level of reliability as vanilla SGD, with both outperforming vanilla SGD for large periods of time. Generally, EEvSGD can be seen to have the most stable performance, followed by SSGD and SGD, with CEvSGD the most unpredictable. The pith of the question is: how can we tackle unpredictability while maintaining superior accuracy and efficiency?

Considering the fact that when CEvSGD results blow up, one usually finds out within the first 50 steps - the most practical idea, from a computational perspective, is to have a stopping criteria in case of extreme values. When this stopping criteria is reached, the model can reset. Assuming an unrealistically unlucky scenario in which CEvSGD blows up five times in a row - implementing this checking criteria only costs us 250 gradient descent steps and some computational efficiency for the first 50 steps. In context, these costs seem minimal and worth pursuing. Additionally, we can make the fitness more agressive in theory to shed off problematic terms quickly and even take more aggressive step sizes. Thi swas not tested but promising are to pursue. This would allow for a stablity that other algorithms in general do not enjoy to the addition of being an aggressive algorithm.

As for SSGD, which is the least computationally expensive method, the issue is in the spikes (clearly visible in Figure 2). This dodgy behavior could possibly be circumvented by deploying implicit methods for the gradient descent update [3].

4.1.4 Standardization

As mentioned previously, we did not get the results we expected after standardization. This would require further analysis which we believe could help performance.

4.2 Conclusion

We learned some very interesting things over the course of writing this paper. The sensitivity of optimization algorithms to hyperparameters, specifically those relating to stepping, was a major takeaway. As far as our methods go, we would have loved to try them out on more non-linear, complex domains - we do believe our EvSGD and SSGD methods would perform well here, and it would be very interesting to see how they fare relative to vanilla SGD.

In the linear case we can make the case that this was worth pursuing even considering the extra time each individual took. Looking at figure 2 we note that it is a log-log graph. While all our methods took 2-3 times the vanilla method we can clearly see that the vanilla took over 3 times the iterations during similar epochs to arrive at a similar accuracy!

– –

GitHub Link: `https://github.com/ysa257/EvSGD`

References

[1] William M Spears. “Crossover or mutation?” In: *Foundations of genetic algorithms*. Vol. 2. Elsevier, 1993, pp. 221–237.

[2] David E Goldberg. “Genetic and evolutionary algorithms come of age”. In: *Communications of the ACM* 37.3 (1994), pp. 113–120.

[3] Panos Toulis and Edoardo M Airoldi. “Implicit stochastic gradient descent for principled estimation with large datasets”. In: *ArXiv e-prints* (2014).

[4] Xiaodong Cui et al. “Evolutionary stochastic gradient descent for optimization of deep neural networks”. In: *arXiv preprint arXiv:1810.06773* (2018).