

Distributed Optimization and Machine Learning

Prof. Mayank Baranwal

Computer Science & Engineering, Electrical Engineering, Mathematics

Indian Institute of Technology Bombay

Week-10

Lecture 36: Algorithms for Distributed Optimization-2

So, let us move to a slightly or a second order variant of it. When I say second order that means just like we made use of momentum in the context of centralized gradient descent and developed heavy ball method or the Nesteros method that had faster convergence rate than the simple gradient descent or vanilla gradient descent. We can design something very similar here as well and we are going to design something called distributed accelerated or distributed aggregated gradient descent feather. It is a second order algorithm. And the idea is I mean so the agents are going to initialize themselves to some x_i^0 right in the beginning because agents they do not know what other agents are initializing to right.

So, everyone would have their own initial. So, this is the initialization step. So, you have x_i^0 s and we define something called y_i^0 which are nothing but gradient of f_i evaluated at x_i^0 . So, we are going to, so agents are going to be initializing two variables now, because they are going to be exchanging two variables.

As we had seen in the context of momentum based methods, we had exchange of not exchange, but we were working with two variables at a time right. So, in this case if you want to use momentum, you are you also have to use the previous gradient information right. And therefore, the you need to work with x both x_i and y_i and y_i is like a proxy for a momentum term ok. So, the way this algorithm works it after any iteration at the end of k iterations. So, you are going to be defining x_i^{k+1} .

So, you have the same similar kind of mixing step, but slightly different variant of it $j+1$ through n $w_{ij} x_j^k$. So, this would have been for a simple first order algorithm, but now that we are working with second order algorithm we also have some step size α times y_i^k ok. And think of y_i^k in the beginning and think of y_i^k as the gradient right. So, and then you also need to update your y_i as. So, you run another consensus on y_i .

Distributed Aggregated Gradient Descent.

(2nd-order algorithm)

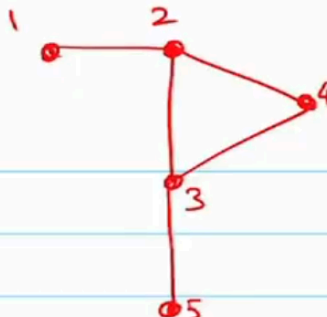
• Initialization-step

$\{x_i(0)\}$ $\{y_i(0) := \nabla f_i(x_i(0))\}$

- $x_i(k+1) = \sum_{j=1}^N w_{ij} x_j(k) - \alpha y_i(k)$
- $y_i(k+1) = \sum_{j=1}^N w_{ij} y_j(k) + \nabla f_i(x_i(k+1)) - \nabla f_i(x_i(k))$

So, w_{ij} $y_j(k)$, then you have a gradient f_i evaluated at $x_i(k+1)$ minus gradient f_i evaluated at $x_i(k)$. This is the algorithm. Is the algorithm clear? So, we are looking at the we are using the previous gradient information as well as the current gradient information and that is where with that basically is your momentum term right. So, you are trying to use that momentum term and then you want to ensure that you want to get the sublated momentum term and in your when you perform a consensus and gradient step essentially you I mean essentially this is your step right where you combine both the mixing as well as the gradient step. So, is this algorithm clear? So, let us look at the, so I have coded this up.

Ex:



$f_i(x) := \frac{1}{2} (x-i)^2$

$\nabla f_i(x) = (x-i)$

$\sum_{i=1}^N f_i \rightarrow x^* = 3$

So, let us look at the implementation of the two algorithms and for the same example. So, let me rewrite sort of redraw the example. So, we are going to be looking at the similar setting. ok and then you have f_i is to be half x minus i whole square. So, gradient of f_i of x that should be x minus i and x star.

So, if you are trying to minimize $i=1$ through n f_i right. So, x star turns out to be what is x star in this case n is equal to 5 right there are x star is 3 ok. So, let us try and see if we can make these algorithms to work all right. So, the first part is just importing the basic library. So, numpy and just for plotting the matplotlib is what we are going to use.

Now we are trying to minimize some of these convex functions of x_i , yeah by the way there is the DGD and the I mean it is needless to say that these are I mean objective functions are supposed to be convex right. I mean we have been working with convex optimization all through. At best we can work with the function that satisfy peer inequality, but not for a general non-convex function. So, this part is fine. So, let us define the graph.

So, if I look at the if I look at the the weighting matrix, what should it look like? I consider the weighting matrix. So, it should be w_{ij} is essentially $\frac{1}{1 + \max(d_i, d_j)}$ right. So, I have defined. So, number of iterations is the number of iterations in that both the algorithms are going to be run for η is a learning rate η we are going to keep it both fixed as well as $\frac{1}{k}$ kind of learning rate is what we are going to see. The other thing that we are going to be dealing with is, so essentially the graph connectivity or the network connectivity right.

So, in this case E essentially is basically your edge set. So, it is going to get a value of 1 if there is an edge otherwise it is going to be 0 right. So, 1 is connected to 2. So, that is why you have an edge between 1 and 2 and vice versa it is an undirected graph. 2 is connected to 3 and 1 and 4.

So, you can see 1 has basically connectivity with 0, 1, 0, 2 and 3 and so on. I mean indexing starts with 0 in Python right. Then I get the degree matrix because in computing WIG I need to make use of the degree information. So, degree matrix is nothing but the sum of row sum or the column sum however you want to see for the edge matrix. So, W is basically $\frac{E_{ij}}{d_i}$ over this this is the formula that we had seen right and likewise you can define W_i right.

So, let us define this graph and if I look if I just print this particular graph or the graph connectivity this is what it looks like. for agent 1 which is connected to agent 2. So, essentially you have 0.75 at the self weight and 0.25 at the weight of 1 and 2 right.

And one thing to note this note here is this particular matrix is W stochastic right, this matrix is W stochastic. So, that part is clear and that is one of the properties that we wanted for consensus to work. Now, the first order DTD algorithm So we initialize the agents to random values. So you can see that `np.`

`random.randn` and it is going to generate 5 random values using a standard normal distribution. And then I am going to be recording all the values of individual agents after each iteration. So, in the first I mean this is the initial value x_{val} I mean that. So, basically it is number of iterations times number of agents. So, for the first iteration it is

simply the x naught and then I am running the algorithm which is every agent I mean I can basically say that w times x is what they are running right.

And once you get this x i k plus half you essentially compute you perform the gradient descent step which is z minus step size times ∇z . So, z naught is a vector of 1, 2, 3, 4, 5. If you can see z naught is essentially 1, 2, 3, 4, 5 because we know that the gradients are essentially x minus i . So, that is why I am using z minus. So, this i is essentially the z naught vector for every agent. So, this is what we are doing here and then we are just record updating the new, we are just basically saving the new value right.

So, let us run this algorithm and let us plot this. So, you can see the agents in the beginning they start at a different value right every agent has its own starting value some are negative 2 point some 2.5 or one of them is 1.2, but eventually all of them they start converging towards 3 though not exactly converge to 3, but at least start converging to 3 right. So, this is this is how it works.

In terms of how quickly it converges, maybe after 200 iteration, it sort of starts showing convergence. It pretty much looks like it has converged beyond this point. Even though it has not converged to exactly to 3, it has converged close to 3, right. So, can we make it faster? It turns out, let us try and see if we can make it faster. So, we run it with a larger learning rate and let us see what happens.

Now you see that even though the convergence is fast the values that they have I mean there is no consensus going on right and this is one of the problem with this decentralized gradient descent algorithm or distributed gradient descent algorithm. So, the convergence is not exact and that is because the two dynamics one of them is the consensus dynamics and then followed by the gradient dynamics. So, the two dynamics kind of sort of I mean interfere with each other. So, if you try and look at the fixed point of this.

So, let me switch to here. So, if you look at the fixed point of this algorithm. So, when does this become 0? Like if I look at the single step of this. So, when this quantity is equal to this quantity and that's right. So, what this says is that essentially if you have a finite step size that means individually every gradient has to be 0 and that is the problem right. at x equal to 3 x^* equal to 3 individually not all gradients are 0 only one of gradients of one of the objective function is 0.

So that means you do not want this dynamics to happen faster like I mean you do not want x_i to be like consensus to happen faster otherwise this term would ensure that like I mean they individually every gradient is and that's why you see if you run the consensus step if you use a very large learning rate you see that everyone has more or less

converged I mean they are trying to converge closer to their private objective like the private optimal value right, not exactly but closer to their private optimal value. And so this is the sort of interplay between the consensus step and the gradient step in the DGD algorithm. And let us try and see if the other algorithm the second order algorithm is any better in that sense. there is no consensus here. So, a suitable learning rate let us say let me decrease the learning rate even further right.

Now, you can see the values are pretty close to each other, but it now takes time to converge right. So, increase the number of iterations as well. So now you see that everyone has closed to 3. So that means the gradient step is not fast enough. So consensus step is, there is consensus as well going on between the agents.

But again, so this is one particular issue with that. So now if I let's say want to use a larger learning rate. and this is where you see this particular issue right. So, if I want to use a larger learning rate, one thing that we said was we probably need to use some kind of n plus 1 or some kind of decaying learning rate right.

So, let me run this. So, now you can see that for the same learning rate the values are like I mean it takes roughly similar amount of time to run basically to settle down, but the values are closer to 3 ok. So, if I decrease the learning rate let us say even further. I mean it has not converged, but you can see the values are pretty close to each other right. So, this is one way to sort of get rid of the difference. So, in Python if you want to do matrix vector product $w^* x$ does element wise product.

So, if you want to do matrix vector product you have to use `at`. So, either you use `at` or there is a command called `np.dot` `matmul` `np`. Yeah, yeah. So, this this does a matrix like the usual matrix vector product that you want otherwise it does the element wise products essentially it would be dot product with every column of the every column of the matrix. So, one thing that we saw was that you may have to use variable learning rate or basically decaying learning rate if you want the convergence to be more exact right.

So, this is so let us let us rerun this define the graph. So, with a small learning with a constant, but small learning rate, you essentially get convergence decent like good enough convergence rate. Now let us see how is the second order algorithm looks like which is, so in second order algorithm the agent is going to be exchanging both x as well as y right and let us say the agent initializes at some x naught. So, this is your x naught then your y is essentially y naught is nothing but the gradient of f y evaluated at x naught right and that is what we are doing and then we are also going to be keeping track of the y variable because agents also run consensus on y or the mixing on y . So, in this case let us say I use a learning rate of 0.

01. So, this is the algorithm right you get new x which is w times x your mixing matrix mixing step and your gradient descent gradient descent step combined and then your new y is basically mixing on y then you basically look at the current grade like gradient evaluated at the new point minus the gradient evaluated at the previous point. So, that is what we are doing and then we are just sort of keeping track of then x and y values. So, let us run this algorithm and let us see how the convergence looks like and you can see it is super fast right. For the same learning rate it converges much faster moreover like you can see that the difference between the two terms is essentially the difference between the x i's So, there is also consensus on this and that is because we are running additional consensus on the y variable as well. So, if you analyze the dynamics of this, I mean there is no sort of interplay between whether you want to do the gradient descent first or the consensus first right.

So, for different learning rates even let us say 0.1, this works. Even for 0.1 kind of learning rate, it works much faster. So, that means decrease the number of iterations otherwise.

So, this is how this algorithm works. So, there is very sort of tight sort of consensus between the algorithms. I mean between the different optimizers or different estimates of the optimizers as well as they converged faster than the previous algorithm right. So, it is much more robust the downside is if x is very large dimensional then you are exchanging two times of that like basically twice the amount of information right. So, you need I mean so there is some sort of in terms of the communication requirement the communication bandwidth requirement is just double of the previous case, but then you get this faster convergence and more robust convergence. okay.