Optimization

Master on Fundamentals of Data science
Universitat de Barcelona

Optimization Exam

Blai Ras

January 20, 2021

1 In your opinion, which is the most important result (lemma, theorem or proposition) of the course. Justify clearly your answer.

For me, the most astonishing lemma/theorem/proposition that we've seen is the Ant-colony heuristic method. Therefore, I think is one of the most important that we've seen because one can be surprised that we can optimize or approach a problem by "imitating" the behavior of ants!

Let me first introduce what an heuristic is. An heuristic is an approach to a problem that employs a practical method that is not guaranteed to be optimal. In other words, it's like a "generalized guide", meaning that it works for multiple types of problems. Unfortunately, this kind of problem-solving approaches do not guarantee feasibility, meaning that our heuristic may fail trying to solve the problem.

The heuristic that I'm going to talk about is Ant-colony. This method is based of the biologic behavior of the ants, specifically, on their pheromone trails, a chemical substance that each ant leaves of the ground and that it can be "read" or "smelled" by other ants. This can be understood better with an example: there's an ant-colony in a point A. Two ants leave the colony with the intention of finding food in a point B. One ant does not reach point B, the food. The other one is more lucky and finds it. Both ants come back to the colony, point A, but the behavior changes: the ant that found the food leaves more amount of pheromone on the ground that the one who did not find the food. Consequently, all the ants that will leave the colony in search of food will be guided by the path with more amount of pheromone, therefore finding the food and surviving.

That's a global idea, but, in reality, when implementing this ACO algorithms, some changes are introduced:

- Simulated ants only leave pheromone on the way back to the nest. Real ants always leave pheromone.
- Simulated ant-colonies evaluate a solution with respect of some quality measure, manipulated by any parameters or constants, while real ants are more simple and receive pheromone reinforcement quickly.

An generalized ACO pseudo-code is the following:

Algorithm 1 Generalized ACO pseudo-code

Initialize solution components S_c, define max. number of ants as N Define f as objective function Define pheromone update function, with the pheromone matrix and p as pheromone evaporation ratio. Define termination criteria (max. number of iteration, convergence, etc) **while** termination criteria not met **do** do
while ! AllAntsFinished **do**
do **for** AllAnts **do** do Construct a possible solution, for example, a vector of fixed length defined by the problem. Add this vector to the solution based on probability *prob*
end for
end while
Perform the pheromone update taking into account the objective function, possible constraints. Compute Global update
end while

In order to go in-depth of some of this steps we can imagine a global ACO problem as a Graph G=(D,P) where D are the set of nodes and P the pheromone trails that unite them.

Let's start with defining the probability of each ant to go from one node to the other:

$$p_{ij}^k(t) = \begin{cases} \frac{\tau_{ij}^\alpha(t)}{\sum_{h \in N_i^{(k)}} \tau_{ij}^\alpha(t)} & \text{if } j \in N_i^{(k)} \\ 0 & \text{if } \notin N_i^{(k)} \end{cases}$$

Where k is a given ant, i is the starting node and j the landing node and τ_{ij} is the pheromone trail link.

Then, of course, we have to define the amount of pheromone an ant leaves behind:

$$\Delta\tau_j^k(t) = C(f(x_i) - f(x_j))$$

Where C is a positive constant. The bigger value of $f(x_i) - f(x_j)$, the bigger amount of pheromone an ant will leave. Then path I_j to I_k will be more appealing for other ants.

If we sum all this amounts we will define the total amount of pheromone left behind:

$$\Delta\tau_j(t) = \sum_{k=1}^{q} \Delta\tau_j^k(t)$$

When all the ants finish the iteration, we update the value of $\tau_j(t+1)$ as:

$$\tau_j(t+1) = (1-p)\tau_j(t) + \Delta\tau_j(t)$$

Where p is the coefficient of evaporation.

This is a global idea of the algorithm, but I would like to remark some notes that must be taken into account:

- We have to handle loops in the graph.
- We have to avoid falling onto "a local maximum or minimum", meaning that ants tend to go to the most close solution and we are not exploring other possible ones!

Real-word applications of this method can be:

- Optimizing a staff of employees of a company in order to get the maximum amount of work done working the less possible amount of hours, respecting some constraints such as not working on holidays, max. 40 hours of work a week, etc. (Scheduling problems)
- Optimizing vehicle routing for package delivery (Routing problems)
- Network routing: how net packages have to be handled and delivered on a router.
- The Knapsack problem, The Sailsman problem, etc.

This proposition may not be as famous/relevant as other ones but I think it's important because it's atypical from other methods seen in class but sometimes it works better as other methods that try to find the exact solution.

2 In your opinion, which is the most important optimization method or algorithm among the ones that have been explained in the course. Justify your answer considering its requirements, constraints, advantages and drawbacks.

For me, Gradient descent algorithms are the most important. Gradient descent tries to find a local minimum of a differentiable function. It tweaks parameters iteratively to find the best solution. A gradient, instead, is a partial derivative of the problem function. In other words, a measure or the "slope" of a function. The higher, the faster we are "learning", but, if its zero, we are not learning anything.

Gradient descents does the following:

$$b = a - \gamma \Delta f(a)$$

Where b is the next position of our "climb" and "a" is our current position. γ is a "waiting factor" and $\Delta f(a)$ is the direction of our gradient. With this function we iteratively move to the minimum of the function step-by-step, archiving this minimization.

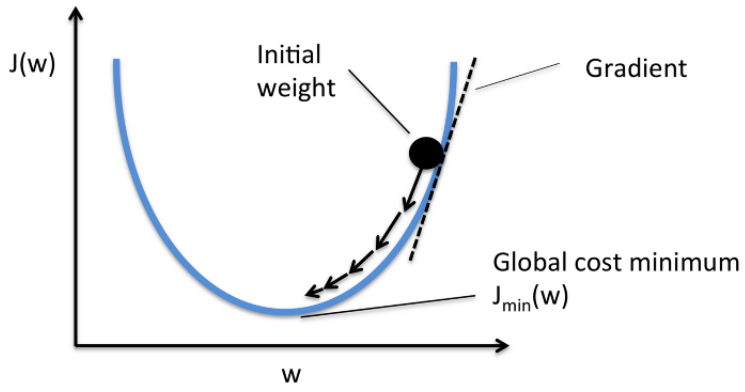

Figure 2.1: Minimization example of a parabola function J(w)

Given that is an iterative method, the learning rate is extremely important. The learning rate in this case is "how big are our steps" when finding the minimum. Therefore, multiple variations of this methods exist in order to ensure the best learning rate on each iteration:

- **Batch Gradient descent**: calculates the error after the training all the samples of our problem. We call this process "an epoch". Advantages: is computationally efficient, gives a stable error gradient and convergence. We are taking a "direct path" towards the minimum. Disadvantages: sometimes this stable error gradient can be a "local" solution (local minima).
- **Stochastic gradient descent**: in this case we are updating the parameteres for each training example one by one. Advantages: it can be faster than the batch gradient descent and we have notion at each iteration of the improvement rate. We are more likely to find the "real" minimum. Disadvantages: we can take incorrect directions due this frequency of updated. Computationally more expensive!
- **mini-batch Gradient descent**: this one combines the two last methods. In this case we are splitting the data into small "batches" and performing the update of each one. Therefore, we are balancing the robustness of the SGD and the efficiency of the batch one. Usually, mini-batches of size 50 to 200 are taken. This size is one of the disadvantages of this method: choosing it may affect the whole minimization and finding the ideal one has to be done through hyperparameters.