Ex .1 - Discovering gradient descent

All code is in `ex1.ipynb` with notation and result.

1. The method of **Least Squares** minimizes the sum of the **squares** of the residuals between the observed targets in the dataset, and the targets predicted by the linear approximation. Check `Problem 1` for graph.
2. **a)** Check `Problem 2` for the code and result.
3. **a)** By fixing b, they adjust the value of a by a very small value each time to decrease the value of SSE. This is equals to considering the partial derivative of SSE with respect to a, namely $\frac{\partial SSE}{\partial a}$. Gradient descent algorithm also consider this value and updates the value of a using the equation, $a = a - lr * \frac{\partial SSE}{\partial a}$, where lr is a small number called learning rate.

   b) We can use the algorithm below.

   ```
   iteration = 0
   generate(a,b)
   loss = 0
   prec = 0.01

   while iteration ≤ 100 and loss ≥ prec:
           a_now = a - lr * partial_derivative_a
           b_now = b - lr * partial_derivative_b
           y_pred = b_now * X + a_now
           loss = SSE
           a = a_now
           b = b_now
           iteration += 1
   ```

   The result is displayed in `Problem 3`.
4. **Batch Gradient Descent**: all the training data is taken into consideration to take a single step. We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters. So that's just one step of gradient descent in one epoch.

   Stochastic Gradient Descent: We consider just one example at a time to take a single step in case the dataset is too large.

   The code and result are displayed in `Problem 4`.

Ex .2 - Gradient descent on big data

Let m be the number of data samples.

1. **a)** Depth: $O(logm + logn)$.

   Work: $O(2n + 2m)$.

   b) $\nabla wF = 2\sum_{i=1}^{m} x_i^T(x_i^T w - y_i)$.

   Depth: $O(logm + logn)$.

   Work: $O((4n + 2m) * log\frac{1}{\epsilon})$.

   c) Gradient descent has a very large depth relatively. The whole dataset is considered in each iteration. When we deal with big data, its parallelization ability is weak. Therefore, it's not very suitable for parallelization and big data.
2. **a)** For stochastic gradient descent, $\nabla wF = 2x_i^T(x_i^T w - y_i)$.

   Depth: $O(logn)$.

   Work: $O(\frac{2n}{\epsilon})$.

   b) In each iteration, there are multiple processors to compute and update the value of w. Therefore, we could encounter the problem of multiple processors writing w at the same time or we can not ensure atomic writing operation.

c) Yes. However, in the context of big data, the time latency to open a lock is very expensive. Therefore, it may not be a viable solution.

d) No. Considering the amount of the data, each F_i takes a very small amount of time to compute. This could decrease the probability of lock problems. Additionally, big data is able to tolerate a partial mistake.

3. **a)** Rows. Because x_i is a column of feature. Therefore, since we use x_i^T a lot, we can store them in rows.

   b) First, we sum up nodes in the same raft. Then sum rafts and send the result to each data center. Finally, we add up results of each data center to the root to present compute the final loss.

   c) The communication cost is great because there are different levels of communication between nodes, rafts, data centers. On the other hand, the amount of information to send is reduced by the parallelized data transportation. The overall cost may be reduced.

4. **a)** Marks the current stage as a barrier stage, where Spark must launch all tasks together. In case of a task failure, instead of only restarting the failed task, Spark will abort the entire stage and relaunch all tasks for this stage. Therefore, I think the strategy in 2.d is very suitable. We can consider each F_i as a barrier stage, which allow all tasks to be launched at the same time. Since only a small number of components of w is calculated in each stage, the stage will be very fast. Even if some lock issue happens during a stage, we can simply rerun the tasks from the current stage with a very small time cost, thus no need for locks.

   b) Tensorflow

Ex .3 - PCA and gradient descent

The objective is to find the directions in which the variance, $E(X, X^T)$, is maximum.

Let w denote the unit vector direction along which the variance is maximum. The variance along this direction is given by:

$\sigma_{w^2}^2 = \frac{1}{n}\sum_i (x_i w)^2 = \frac{1}{n}(xw)^T(xw) = \frac{1}{n}w^T x^T x w = w^T \frac{x^T x}{n} w = w^T v w.$

We can take gradients w.r.t to w and set it zero to find the optimum values.

$L(w, \lambda) = \sigma_{w^2}^2 - \lambda(w^T w - 1),$

$\frac{\partial L}{\partial \lambda} = w^T w - 1,$

$\frac{\partial L}{\partial w} = 2vw - 2\lambda w.$

We can solve minimizing L using gradient descent because the above function is concave. Its Hessian matrix is:

$$\begin{bmatrix} 0 & -2w^T \\ -2w & 2v - 2\lambda I \end{bmatrix}$$

The determinant of this Hessian can be computed as:

$$det \begin{bmatrix} A & B \\ C & D \end{bmatrix} = det|D|deg|A - BD^{-1}C|.$$

The determinant is calculated as the expression:

$-det(2v - 2\lambda I)det(2w^T(2v - 2\lambda I)^{-1}2w),$

which is alway negative. Therefore, it will converge to maximum.