CME 323: Distributed Algorithms and Optimization

Instructor: Reza Zadeh (rezab@stanford.edu)
TA: Alex Yang (yangzj@stanford.edu)

HW#2 – Due Thursday May 4 (on Gradescope)

1. **List Prefix Sums** As described in class, List Prefix Sums is the task of determining the sum of all the elements before each element in a list. Let us consider the following simple variation.

   - Select each element from the list randomly and independently with probability $1/\log n$ and add it to a set S. Add the head of the list to this set, and mark all these elements in the list.
   - Start from each element $s \in S$, and in parallel traverse the lists until you find the next element in S (by detecting the mark) or the end of the list. For $s \in S$, call this element found in this way $\texttt{next}(s)$. While traversing, calculate the sum from s to $\texttt{next}(s)$ (inclusive of s but exclusive of $\texttt{next}(s)$), and call this $\texttt{sum}(s)$.
   - Create a list by linking each $s \in S$ to $\texttt{next}(s)$ and with each node having weight $\texttt{sum}(s)$.
   - Compute the List Prefix Sums on this list using pointer jumping. Call the result $\texttt{prefixsum}(s)$.
   - Go back to the original list, and again traverse from each s to $\texttt{next}(s)$ starting with the value $\texttt{prefixsum}(s)$ and adding the value at each node to a running sum and writing this into the node. Now all elements in the list should have the correct prefix sum.

   Analyze the work and depth of this algorithm. These should both be given with high probability bounds.

Solution

Proposition. *With high probability $|S| = O(n/\log n)$.*

Proof. We will show that the size of S is $O(n/\log n)$ with high probability using a Chernoff bound. Associate indicator variable X_i with the i-th element of the initial list to indicate whether it has been selected for inclusion in S. Since we sample each element with probability $1/\log n$, we see that X_i's are Bernoulli distributed where

$$X_i = \begin{cases} 1 & \text{with probability } 1/\log n, \\ 0 & \text{otherwise.} \end{cases}$$

Clearly, the X_i are independent. Since our X_i's are indicators, then their probability equals their expectation.[1] By linearity of expectations,

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} \frac{1}{\log n} = \frac{n}{\log n}.$$

[1]This follows trivially, since $\mathbb{E}[X_i] = 1 \cdot \Pr(X_i = 1) + 0 \cdot \Pr(X_i = 0) = \Pr(X_i = 1)$.

That is, in expectation $|S| = n/\log n$. Recall Chernoff's Bound, for $\delta \in (0,1)$,

$$\Pr\left(X > (1+\delta)\mu\right) < e^{-\delta^2\mu/3}.$$

We evaluate for $\delta = 1/2$ and $\mu = n/\log n$,

$$\begin{aligned}\Pr(X > 3n/(2\log n)) &< e^{\frac{-n}{12\log n}} \\ &< e^{-\log n} = \frac{1}{n}\end{aligned}$$

We have used the fact that for sufficiently large n, $\frac{n}{12\log n} > \log n$. Note that the analysis can be made more tight, but this probability bound will suffice to say that with high probability, $|S| = O(n/\log n)$. □

Total Work We see that steps a, b, c and e all require $O(n)$ work in the worst case. In step d, we use pointer jumping to perform prefix sums of our linked list. This algorithm, on a list of size n, requires $O(n \log n)$ work and $O(\log n)$ depth.[2] Therefore, step d requires $|S|\log|S|$ work. With high probability:

$$|S|\log|S| \le \frac{3n}{2\log n}\log\left(\frac{3n}{2\log n}\right) = O(n).$$

Total work, then, is $O(n)$.

Total Depth Now we compute the total depth. Steps a and c have depth $O(1)$ because we may do all nodes in parallel. As we stated before, computing prefix sums using pointer jumping on linked lists takes depth $O(\log|S|) = O(\log(n/\log n)$. In steps b and e we need to traverse all elements in the original list between elements in S, so the depth of these steps will be the maximum length between elements in S. We will show with high probability this maximum length will be less than $4\log^2 n$. If this claim holds, then total depth for the algorithm will be $O(\log^2 n)$ with high probability.

Proposition. *The maximum length between elements in S is less than $O(\log^2 n)$ with high probability.*

Proof. Consider dividing the original list into chunks of size $2\log^2 n$ (so there will be $n/(2\log^2 n)$ chunks). For every element in the original list, the probability it is not chosen to S is $(1 - 1/\log n)$, independently. The probability that all of the elements in a particular chunk, c_i, of the original list are not in S is given by:

$$\begin{aligned}\Pr(e \notin S, \forall e \in c_i) &= (1 - 1/\log n)^{2\log^2 n} \\ &= ((1 - 1/\log n)^{\log n})^{2\log n} \\ &\le (1/e)^{2\log n} < \frac{1}{n^2}\end{aligned}$$

[2]Please see the Wikipedia page on pointer jumping and/or `http://wwwmayr.informatik.tu-muenchen.de/lehre/2013WS/pa/split/sub-Prefix-Sum-single.pdf` for a discussion on pointer jumping for prefix sums.

By union bound, we know the probability that all the chunks have at least one element in S is $\leq n/(2\log^2 n) \cdot 1/n^2 \leq 1/n$. So with high probability the maximum length between elements in S is bounded by $4\log^2 n$ (the maximum interval between points in consecutive chunks). Thus, the depth of steps b and e is $O(\log^2 n)$ with high probability, and that is the total depth of the algorithm as well. $\square$

2. **Shortest Path for Weighted Directed Graph** Consider a directed graph $G = (V, E)$ with non-negative weights $\omega : E \to \mathbb{R}^+$. The task is to develop an algorithm to efficiently find the shortest paths from the source $s \in V$ to any other vertex $v \in V$, i.e.

$$p^*(v) = \text{argmin}_{p\,\text{valid}} \sum_{i=1}^{n_p} \omega_i(u_i, u_{i+1}),$$

where a valid path from source s to v is a sequence $p = (u_0, u_1, \ldots, u_{n_p})$ satisfying:

- $u_0 = s$;
- $u_{n_p} = v$;
- $(u_i, u_{i+1}) \in E$ for $i = 0, \ldots, n_p$.

Now assume that the operation of finding neighbors of a vertex can be performed in constant time and constant depth (both $O(1)$). Design an algorithm that achieves $O(nm)$ work and $O(n \log n)$ depth. ($n = |V|$ and $m = |E|$.)

Solution First, we present the following algorithm about solving Shortest Path Problem:

```
function BellmanFord($G$, $s$, $\omega$)
    $k \leftarrow 0$
    $D_k \leftarrow \{v \mapsto \infty : v \in V \backslash \{s\}\} \cup \{s \mapsto 0\}$
    $P_k \leftarrow \{v \mapsto null : v \in V\}$
    while $k == 0 \;||\; D_k \neq D_{k-1}$ do
        for $v \in V$ (parallel) do
            $D_{k+1}[v] \leftarrow \min\left(D_k[v], \min_{u \in N_G^-(v)}\{D_k[u] + \omega(u, v)\}\right)$
            $P_{k+1}[v] \leftarrow \text{argmin}_{u \in N_G^-(v)}\{D_k[u] + \omega(u, v)\}$
        end for
        $k \leftarrow k + 1$
    end while
    return $D_k$
end function
```

Here, $N_G^-(v)$ indicates the in-neighbors of vertex v.

Then we prove the correctness of the algorithm.

Theorem (Correctness of Bellman-Ford). *Given a directed weighted graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}^+$, and a source $s \in V$, the Bellman-Ford algorithm returns a shortest-path tree for all vertices reachable from s.*

Proof. By induction on the number of edges n in a path. The base case is correct since the initial distance to the source s is set to zero. For all $v \in V$, on each step a shortest s-v path with up to k edges must consist of a shortest s-u path with up to $k-1$ edges followed by a single edge (u, v). Therefore, if we take the minimum of these, we get the overall shortest path with up to k edges. For the source, the self-edge will maintain the distance of zero. The algorithm can only proceed to $|V|$ rounds if there is a reachable negative-weight cycle. Otherwise, a shortest path to every vertex v is simple and can consist of at most $|V|$ vertices and hence $|V| - 1$ edges. $\square$

Now, let's analyze the work and depth of this algorithm: the algorithm converges in at most n steps, and for each step, we find the minimum element of a list for each node in parallel, which takes $O(|N_G^-(u)|)$ work and $O(\log |N_G^-(v)|)$ depth.

$$W_{BF} = O\left(n * \sum v \in V \left|N_G^-(u)\right|\right) = O(nm),$$
$$D_{BF} = O\left(n * \max_{v\in V} \log \left|N_G^-(v)\right|\right) = O(n \log n)$$

3. **Singular Value Decomposition for SPSD Matrices** Given a symmetric positive definite matrix $A \in \mathbb{R}^{n\times n}$, in this problem we explore how to efficiently compute its singular value decomposition $A = U^T SU$, where U is orthogonal matrix, S is diagonal. Here we assume the singular values satisfy $\lambda_1 > \lambda_2 > \cdots > \lambda_n$.

   (a) Instead of directly performing QR-iteration on A, we want to first convert matrix A to a symmetric tridiagonal matrix using Householder reflection, i.e. constructing an orthogonal matrix Q_0, s.t. $T = Q_0^T AQ_0$ is symmetric tridiagonal. Give a parallel algorithm to solve this problem, and analyze its work and depth. (Hint: Householder reflection $H = I_n - 2\frac{(e_\alpha - e_\beta)(e_\alpha - e_\beta)^T}{(e_\alpha - e_\beta)^T(e_\alpha - e_\beta)}$ projects e_α to e_β where $e_\alpha = \alpha/||\alpha||_2$ and $e_\beta = \beta/||\beta||_2$.)

   Solution

```
function HouseholderTridiagonalization($A$)
    $T, Q_0 \leftarrow A, I$
    for $k = 1$ to $n-2$ do
        $\alpha \leftarrow -\mathrm{sgn}(A[k+1, k]) \cdot \sqrt{\sum_{j=k+1}^{n} A[j, k]^2}$
        $r \leftarrow \sqrt{\frac{1}{2}(\alpha^2 - A[k+1, k] \cdot \alpha)}$
        $v \leftarrow \frac{1}{2r} A[k+2 : n, k] - \frac{\alpha}{2r} e_1$
        $P \leftarrow I - 2vv^T$
        $T, Q_0 \leftarrow PTP, PQ_0$
    end for
end function
```

In the above algorithm, we sequentially apply Householder Transformation to zero out off-tridiagonal elements for each row and column i. While the sequential nature of this step precludes parallelization, the transformations themselves involve matrix-vector products and element-wise matrix operations. These can be parallelized to achieve a depth complexity of $O(\log n)$.

$$W(n) = \sum_{k=1}^{n} O(k) + O(nk) = O(n^3)$$

$$D(n) = \sum_{k=1}^{n} O(1) + O(\log k) = O(n\log n).$$

(b) Now we perform QR-iteration on symmetric tridiagonal matrix T to achieve SVD for T using Givens rotation. Obviously, if we get the SVD of T, i.e. $S = \tilde{Q}^T T \tilde{Q}$, then we have $S = (Q_0\tilde{Q})^T A Q_0 \tilde{Q}$, which is the SVD of A. Now, let's explore the following algorithm:

```
function QR-Iteration for symmetric tridiagonal matrix($\boldsymbol{s}[1 \dots n]$)
    Let $Q_0 \leftarrow I_n$, $T_0 \leftarrow T$
    for $t = 1$ to $T$ do
        Let $Q_t \leftarrow Q_{t-1}$, $T_t \leftarrow T_{t-1}$
        for $k = 1$ to $n - 1$ do
            Let $\alpha \leftarrow T_t[k : k+1, k]$, $(c, s)^T \leftarrow \alpha/||\alpha||_2$
            $G_k \leftarrow$ Givens$(k, c, s)$
            $T_t \leftarrow G_k * T_t * G_k^T$
            $Q_t \leftarrow G_k * Q_t$
        end for
    end for
    return $T_T$, $Q_T$
end function
```

Here, the matrix representation of Givens Rotation is

$$\text{Givens}(k, c, s) = \begin{bmatrix} I_{k-1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \begin{bmatrix} c & s \\ -s & c \end{bmatrix} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & I_{n-k-1} \end{bmatrix}.$$

Analyze the work and depth of this algorithm, then compare it with part (a). Write a few sentences about your findings.

Solution The Givens rotation algorithm for QR iteration on a symmetric tridiagonal matrix involves successive application of Givens rotations to zero out the off-diagonal elements. For each rotation, targeting a specific off-diagonal element, only two rows and columns are modified, leading to a computational complexity of $O(1)$ for each rotation. With $n-1$ rotations needed to process all subdiagonal elements once per iteration, the total work per iteration is $O(n)$. Similarly, the depth of this algorithm is also $O(n)$ as the algorithm works sequentially. In our algorithm, we assume T iterations are enough for convergence, thus the total depth and work are:

$$D(n) = O(Tn)$$

$$W(n) = O(Tn)$$

4. **Stochastic Gradient Descent**

   (a) In class we proved that gradient descent on L-smooth functions is guaranteed to decrease the function value at each iteration. Stochastic gradient descent, on the other hand, does not have the same guarantee. Provide an example where stochastic gradient descent does not produce a descent step. Specifically, find a function $f(x) = \sum_{i=1}^{m} f_i(x)$, and an iterate x_0 such that for all step sizes, there exist i such that $f(x_1) > f(x_0)$ (where $x+1 := x_0 - \alpha \nabla f_i(x)$).

   Solution Consider $f(x) = \frac{1}{2}(f_1(x)+f_2(x))$ where $f_1(x) = \frac{1}{2}(x-2)^2$ and $f_2(x) = \frac{1}{2}(x+1)^2$. Suppose $x_0 = 0$ and we sample f_2 first, then $x_1 = x_0 - \gamma(x_0+1) = -\gamma$. So regardless of how we choose $\gamma > 0$, $f(x_1) > f(x_0)$, so this would not be a descent step.

   (b) This exercise will guide you through the convergence proof of SGD. As a reminder, we are proving that if there exists a constant G such that $\mathbb{E}[\|\nabla f_i(x)\|^2] \leq G^2$ and $f(x)$ is μ-strongly convex. Then, with step-sizes $\gamma_k = \frac{1}{\mu k}$, we have

$$\mathbb{E}[\|x_k - x_*\|^2] \leq \frac{\max\{\|x_1 - x_*\|^2, \frac{G^2}{\mu^2}\}}{k}.$$

   - Using strong convexity, prove that

$$\langle \nabla f(x_k) - \nabla f(x_*), x_k - x_* \rangle = \langle \nabla f(x_k), x_k - x_* \rangle \geq \mu \|x_k - x*\|^2$$

     Solution Because f is μ-strongly convex, then we have that

$$f(x^*) - f(x_k) \geq \langle \nabla f(x_k), x^* - x_k \rangle + \frac{\mu}{2} \|x_k - x^*\|^2$$
$$f(x_k) - f(x^*) \geq \langle \nabla f(x^*), x_k - x^* \rangle + \frac{\mu}{2} \|x_k - x^*\|^2$$

     Combining the inequalities:

$$\langle \nabla f(x_k) - \nabla f(x_*), x_k - x_* \rangle = \langle \nabla f(x_k), x_k - x_* \rangle \geq \mu \|x_k - x*\|^2$$

   - Apply the previous step, to express $\mathbb{E}[\|x_{k+1} - x_*\|^2]$ in terms of $\mathbb{E}[\|x_k - x_*\|^2]$, γ_k, G, and μ.

Solution Expanding the norm,

$$\begin{aligned}\mathbb{E}(\|x_{k+1}-x^*\|^2) &= \mathbb{E}(\|x_K-\gamma_k g_k - x^*\|^2)\\ &= \mathbb{E}(\|x_k-x^*\|^2) - 2\gamma_k\mathbb{E}(\langle g_k, x_k-x^*\rangle) + \gamma_k^2\mathbb{E}(\|g_k\|^2)\\ &\le \mathbb{E}(\|x_k-x^*\|^2) - 2\gamma_k\mathbb{E}(\langle \nabla f(x_k), x_k-x^*\rangle) + \gamma_k^2 G^2)\\ &\le \mathbb{E}(\|x_k-x^*\|^2) - 2\gamma_k\mu\mathbb{E}(\|x_k-x^*\|^2) + \gamma_k^2 G^2)\end{aligned}$$

- Prove the convergence of SGD using induction. It is clear that the base case, which states
$$\|x_1-x^*\|^2 \le \max\{\|x_1-x^*\|^2, \frac{G^2}{\mu^2}\}$$
is true. Now assume that statement holds until iteration k, we need to show that it holds for $k+1$. From the previous bullet point we have that
$$\begin{aligned}\mathbb{E}(\|x_{k+1}-x^*\|^2) &\le \left(1-\frac{2}{k}\right)\mathbb{E}(\|x_k-x^*\|^2) + \frac{1}{\mu^2 k^2}G^2\\ &\le \left(1-\frac{2}{k}\right)\frac{\max\{\|x_k-x^*\|^2, \frac{G^2}{\mu^2}\}}{k} + \frac{\max\{\|x_k-x^*\|^2, \frac{G^2}{\mu^2}\}}{k^2}\\ &\le \left(\frac{1}{k}-\frac{1}{k^2}\right)\max\{\|x_k-x^*\|^2, \frac{G^2}{\mu^2}\}\\ &\le \frac{\max\{\|x_k-x^*\|^2, \frac{G^2}{\mu^2}\}}{k+1}.\end{aligned}$$

5. **HOGWILD!** This exercise will provide examples applying the main theorem of HOGWILD!. Recall that in HOGWILD!, the objective function we want to minimize is :
$$f(x) = \sum_{e\in E} f_e(x_e)$$
where we define the hyperedge e to be the subset of variables that f_e depends on. Figure **??** depicts such a graph. Then, if we denote the average degree of the conflict graph as $\overline{\Delta}_C$, convergence is still guaranteed if the core delay is less than $\tau \le \frac{n}{2\overline{\Delta}_C}$ (i.e., no more than τ samples are being processed while a core is processing one).

   - **Graph Cuts** In graph cuts problems, we are given a sparse matrix W which indexes similarity between node. We want to match each node to a list of D classes i.e., we want assign a vector $x_i \in \{v \in \mathbb{R}^D | \sum_{j=1}^D v_j = 1, v_j \ge 0\}$ that solve the following optimization problem.
$$\begin{aligned}&\underset{x}{\text{minimize}} && \sum_{(u,v)\in E} w_{uv}\|x_u - x_v\|_1\\ &\text{subject to} && x_u \in \{v\in\mathbb{R}^D | \sum_{j=1}^D v_j = 1, v_j \ge 0\}.\end{aligned} \tag{1}$$
Prove that
$$\frac{\overline{\Delta}_C}{n} = \mathcal{O}\,(\text{Avg. deg.})$$

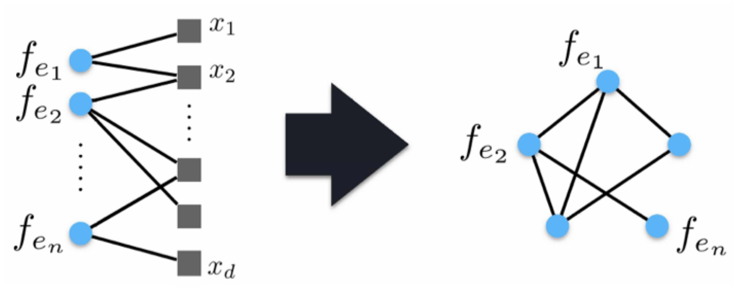

Figure 1: The function-variable and conflict graph for sparse functions.

Solution Since a function $f_{u,v}$ conflicts with all other functions involving u and v, the total number of conflicts is

$$\begin{aligned}
\sum_{(u,v)\in E} deg(u) + deg(v) - 2 &= -2|E| + \sum_{(u,v)\in E} deg(u) + \sum_{(u,v)\in E} deg(v) \\
&= -2|E| + \sum_{u\in V} \sum_{v\in N(u)} deg(u) \\
&= -2|E| + \sum_{u\in V} deg^2(u) \\
&< \sum_{u\in V} deg^2(u) \\
&\leq \left(\sum_{u\in V} deg(u) \right)^2 \\
&= (2m)^2
\end{aligned}$$

So $\overline{\Delta}_C \leq 4m$ and so $\frac{\overline{\Delta}_C}{n} = \mathcal{O}\,$(Avg. deg.).

6. Implement logistic regression using tensorflow. Use the following code to generate train and test data. Note that we have set seed (using "random_state=42"). Use cross-entropy loss and gradient descent optimizer with a learning rate of 0.01. Use batch_size of 100, and run for 500 steps. Report the accuracy on test set.

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Generate data
X_data, y_data = make_classification(n_samples=200, n_features=2,
n_redundant=0, random_state=42)

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_data, y_data,
test_size=0.2, random_state=42)
```

```
# Plot training data
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train)
plt.show()
```

Solution

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np

# Generate data
X_data, y_data = make_classification(n_samples=200, n_features=2,
                                     n_redundant=0, random_state=42)

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_data, y_data,
test_size=0.2, random_state=42)

# Plot training data
#plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train)
#plt.show()

y_train = y_train.reshape((-1,1))
y_test = y_test.reshape((-1,1))

# Define parameters
learning_rate = 0.01
batch_size = 100
num_steps=500
n_samples=X_train.shape[0]

# Define placeholders for input
X = tf.placeholder(tf.float32, shape=[None, 2])
y = tf.placeholder(tf.float32, shape=[None, 1])

# Define variables to be learned
W = tf.get_variable("weights", (2,1), initializer =
 tf.random_normal_initializer())
b = tf.get_variable("bias", (1,), initializer =
tf.constant_initializer(0.0))
y_pred = tf.matmul(X,W)+b
loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=
y_pred, labels=y))

# Define optimizer
opt = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

with tf.Session() as sess:
    #Initialize Variables in graph
```

```
sess.run(tf.global_variables_initializer())
for _ in range(num_steps):
    # Select random minibatch
    indices = np.random.choice(n_samples, batch_size)
    X_batch, y_batch = X_train[indices,:], y_train[indices]
    # Do gradient descent step
    _, loss_val = sess.run([opt, loss], feed_dict={X: X_batch,
    y: y_batch})
    print(loss_val)
# Test model
correct_prediction = tf.equal(tf.argmax(y_pred, 1), tf.argmax(y, 1))
# Calculate accuracy
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print("Accuracy:", accuracy.eval({X: X_test, y: y_test}))
```