Problem Set 3: Accelerating Convergence and Sparsity

CS4787/5777 — Principles of Large-Scale ML Systems

This problem set is assigned on October 2, 2023 and is due about two weeks later on October 18, 2023 at 11:59PM. You may work in groups of up to four students. Submit your solutions on Gradescope.

Problem 1: Minibatching By Sampling Without Replacement.

One variant of minibatched SGD which we discussed in class is *random reshuffling*. Here, rather than selecting a new minibatch by random sampling at each iteration, the algorithm shuffles *the entire dataset* and then produces samples by just running linearly through the dataset in that order. Once it has used the whole dataset (which is a period we call one *epoch*), then it re-shuffles the data for the next pass. Concretely, this corresponds to the following algorithm.

Algorithm 1 Minibatched stochastic gradient descent using random reshuffling

input: learning rate α, minibatch size B where B divides n evenly, initial parameters $w_{0,0}$
input: total number of epochs (passes through the dataset) K
for $k = 0$ **to** $K - 1$ **do**
  sample ς uniformly at random from the set of permutations of $\{1, \ldots, n\}$
    (that is, ς is a bijection from $\{1, \ldots, n\}$ to $\{1, \ldots, n\}$)
  for $t = 0$ **to** $n/B - 1$ **do**
    update model $w_{k,t+1} \leftarrow w_{k,t} - \alpha_t \cdot \frac{1}{B} \sum_{i=1}^{B} \nabla f_{\sigma(i+Bt)}(w_{k,t})$.
  end for
  $w_{k+1,0} \leftarrow w_{k,n/B}$
end for

1.(a) Suppose that we are in a very simple regime where $d = 1$, $n = 2$, and our two component loss functions are:

$$f_1(w) = \frac{1}{2}w^2 + w \qquad\qquad f_2(w) = \frac{1}{2}w^2 - w,$$

where the total loss is $f(w) = \frac{1}{2}(f_1(w) + f_2(w))$. Suppose that we initialize at $w_0 = 1$, and run with step size $0 < \alpha < 1/2$. Find an exact expression for the expected loss of *with-replacement stochastic gradient descent with batch size* 1 after K epochs (that is, after $2K$ total iterations) in terms of K and α.

1.(b) Using your exact expression for the expected value of the loss f from 1(a), show that no matter what α in $(0, 1/2)$ is chosen, the expected value of the loss after K epochs on this task will be

$$\mathbf{E}[f] = \Omega\left(\frac{1}{K}\right).$$

That is, show that there exists a constant $c > 0$, such that no matter what $0 < \alpha < 1/2$ is chosen, $\mathbf{E}[f] \geq \frac{c}{K}$.

1.(c) Now find an exact expression for the expected loss of random reshuffling on this task after K epochs.

1.(d) Using your exact expression for the expected value of the loss f from 1(c), show that there we can choose α as a function of K (the total number of epochs we're going to run) such that the expected value of the loss after K epochs on this task will be

$$\mathbf{E}[f] = \mathcal{O}\left(\frac{1}{K^2}\right).$$

(*Hint:* it is possible to show that for some α as a function of K, $\mathbf{E}[f] \leq \frac{C \log(K)^\gamma}{K^3}$ for some constants C and γ. You are only asked here to show the weaker result that it's $\mathcal{O}(K^{-2})$.)

Problem 2: Understanding Nesterov Momentum. In class, we did an analysis of the convergence of the heavy ball method (Polyak momentum) on quadratic losses. Another popular momentum scheme is Nesterov momentum. The Nesterov momentum step for gradient descent is given by

$$\begin{aligned} v_{t+1} &= w_t - \alpha \nabla f(w_t) \\ w_{t+1} &= v_{t+1} + \beta(v_{t+1} - v_t). \end{aligned}$$

(You can observe that this is very close to gradient descent, but using an exponential moving average of the gradient in place of the gradient.) In this problem, we will try to develop a better understanding of why Nesterov momentum works.

2(a). Suppose that f is the quadratic scalar function

$$f(w) = \frac{\gamma}{2} w^2.$$

Show that the value of v after running t timesteps of Nesterov momentum is determined by the matrix equation

$$\begin{bmatrix} v_{t+1} \\ v_t \end{bmatrix} = \begin{bmatrix} (1+\beta)(1-\alpha\gamma) & -\beta(1-\alpha\gamma) \\ 1 & 0 \end{bmatrix}^t \begin{bmatrix} v_1 \\ v_0 \end{bmatrix}.$$

2(b). Nesterov recommends that the step size hyperparameter α and the momentum hyperparameter, β, be set as

$$\alpha = \frac{1}{L} \qquad \text{and} \qquad \beta = \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1},$$

where $\kappa = L/\mu$ is the condition number of the problem we are trying to solve, L is an upper bound on the largest eigenvalue of a second derivative matrix of the objective (a.k.a. the Lipschitz constant) and μ is a lower bound on the largest eigenvalue of a second derivative matrix of the objective (i.e. the strong convexity constant). Show that, if we use this assignment of α and β, the eigenvalues of the matrix in the above expression are

$$\lambda = \frac{\left(1 - \frac{\gamma}{L}\right)\sqrt{\kappa} \pm \sqrt{\left(1 - \frac{\gamma}{L}\right)\left(1 - \frac{\gamma}{\mu}\right)}}{\sqrt{\kappa} + 1}.$$

(Hint: remember that the eigenvalues of a 2×2 matrix A satisfy the characteristic polynomial $0 = \lambda^2 - \mathrm{trace}(A)\lambda + \det(A)$.)

2(c). Show that, as long as $\mu \leq \gamma \leq L$

$$|\lambda|^2 = \left(1 - \frac{\gamma}{L}\right) \cdot \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}$$

and using this, show that for any γ that satisfies $\mu \leq \gamma \leq L$,

$$|\lambda| \leq 1 - \frac{1}{\sqrt{\kappa}}.$$

(*Hint:* Remember that the determinant of a matrix is the product of its eigenvalues.)

2(d). Now show that, if f is the quadratic function $f(w) = \frac{1}{2} w^T A w$ and A is a symmetric matrix with eigenvalues bounded from below by μ and above by L, and we use Nesterov's suggested hyperparameters as described above, the effect of running t steps will be to get

$$\begin{bmatrix} v_{t+1} \\ v_t \end{bmatrix} = M^t \begin{bmatrix} v_1 \\ v_0 \end{bmatrix}$$

for a matrix M, all the eigenvalues of which have

$$|\lambda| \leq 1 - \frac{1}{\sqrt{\kappa}}.$$

Epilogue. The result you proved here shows that

$$|v_t| = \mathcal{O}\left(\left(1 - \frac{1}{\sqrt{\kappa}}\right)^t\right),$$

where the big-$\mathcal{O}$ notation hides factors that can depend on the initialization of the algorithm and the eigenvectors of the matrix. This shows that, for this simple quadratic objective, we're observing the same replace-κ-with-$\sqrt{\kappa}$ effect that we got from Polyak momentum. The main difference between this and Polyak momentum is that for Nesterov's momentum, this convergence rate also holds for any strongly convex, Lipschitz continuous function, and not just quadratics. Despite the fact that its theory is based in strongly convex gradient descent, Nesterov momentum has been used with SGD to great effect for deep learning (e.g. in the famous paper "On the importance of initialization and momentum in deep learning").

Problem 3: Dimension Reduction. This problem is intended to both explore dimension reduction and give you some more experience with computing the number of floating point operations used in an ML algorithm.

Suppose that you have a training dataset $\mathcal{D}$ of n examples $(x_i, y_i) \in \mathbb{R}^d \times \{-1, 1\}$, for $i = \{1, \ldots, n\}$. Suppose that the examples x_i are stored as an array of dense vectors in $\mathbb{R}^d$ and the labels are stored similarly as an array in $\{-1, 1\}^n$. You want to use this dataset to train a k-nearest-neighbors classifier, with $k = 1$ (i.e. a nearest-neighbor classifier). Recall (from your previous machine learning course) that a nearest neighbor predictor is given for $x \in \mathbb{R}^d$ by

$$h(x) = y_{\arg\min_i \|x - x_i\|_2}$$

where $\|\cdot\|_2$ denotes the ordinary Euclidean ℓ_2 norm.

3(a). Write down pseudocode for how you could compute a prediction $h(x)$ for a test example $x \in \mathbb{R}^d$. Assume that you do not have any special datastructures or indexes to support efficient k-NN, and are just using the naive algorithm (that computes distances between x and x_i for each $i \in \{1, \ldots, n\}$). *(For this part, you may choose to either use numpy code, or write everything explicitly in terms of python for-loops, as long as you feel confident you can use your answer here to work out 2(b) later.)*

3(b). Now suppose that you want to use your algorithm to make predictions for a test set of m examples $\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \in \mathbb{R}^d$. How many floating point operations would your algorithm in 1(a) use to make predictions for all m examples? (Give your answer in terms of m, n, and d.)

3(c). Now suppose that the training dataset $\mathcal{D}$ is sparse: that is, each x_i is represented instead as a sparse vector (using the standard way to store a sparse vector with two arrays: an index array and a values array). Write pseudocode *using only for/while loops (no numpy calls)* that shows how you could compute a prediction $h(x)$ for a test example $x \in \mathbb{R}^d$ stored as an ordinary dense vector.

3(d). Now suppose that you want to use your algorithm to make predictions for a test set of m examples $\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \in \mathbb{R}^d$. How many floating point operations would your algorithm in 1(c) use to make predictions for all m examples? Suppose that the training set $(x_1, \ldots, x_n)$ has average density $p \in (0, 1)$, and give your answer in terms of m, n, d, and p. Note that integer operations, such as indexing into arrays, do not count as floating point operations.