

Lecture 10: Dimensionality Reduction and Sparsity.

CS4787 — Principles of Large-Scale Machine Learning Systems

A quick spill-over from last time. **Stochastic Variance-Reduced Gradient.**

Main idea: mix a lot of stochastic gradient samples with a small amount of full (exact) gradients to cut noise.

Algorithm 1 SVRG: Stochastic Variance Reduced Gradient

given: loss function samples f_1, \dots, f_n , step size α , full-gradient update frequency m
given: total number of outer loop iterations K , initial parameters \tilde{w}_1
for $k = 1$ **to** K **do**
 $\tilde{g}_k \leftarrow \nabla f(\tilde{w}_k) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\tilde{w}_k)$
 $w_{k,0} \leftarrow \tilde{w}_k$
 for $t = 1$ **to** m **do**
 select i_t **at random from** $\{1, \dots, n\}$
 $w_{k,t} \leftarrow w_{k,t-1} - \alpha (\nabla f_{i_t}(w_{k,t-1}) - \nabla f_{i_t}(\tilde{w}_k) + \tilde{g}_k)$
 end for
 $\tilde{w}_{k+1} \leftarrow w_{k,m}$ (there is another “option II” way to update the outer iterate \tilde{w}_{k+1} , but it’s not used in practice)
end for
return \tilde{w}_{K+1}

Why the weird update step? We’re using this new term $\nabla f_{i_t}(w_{k,t-1}) - \nabla f_{i_t}(\tilde{w}_k) + \tilde{g}_k$ in place of the gradient. Does this make sense? Well, notice that in expectation (where all the expectations we’re taking here will be conditioned on $w_{k,t-1}$ and \tilde{w}_k), we have

$$\mathbf{E} [\nabla f_{i_t}(w_{k,t}) - \nabla f_{i_t}(\tilde{w}_k) + \tilde{g}_k] = \nabla f(w_{k,t}) - \nabla f(\tilde{w}_k) + \tilde{g}_k = \nabla f(w_{k,t})$$

so this is actually an unbiased estimator of the gradient, just like we have in SGD. The natural next question to ask is: what is the variance of this estimator?

$$\begin{aligned} \mathbf{E} \left[\left\| (\nabla f_{i_t}(w_{k,t}) - \nabla f_{i_t}(\tilde{w}_k) + \tilde{g}_k) - \nabla f(w_{k,t}) \right\|^2 \right] &= \mathbf{E} \left[\left\| \nabla f_{i_t}(w_{k,t}) - \nabla f_{i_t}(\tilde{w}_k) - \mathbf{E} [\nabla f_{i_t}(w_{k,t}) - \nabla f_{i_t}(\tilde{w}_k)] \right\|^2 \right] \\ &\leq \mathbf{E} \left[\left\| \nabla f_{i_t}(w_{k,t}) - \nabla f_{i_t}(\tilde{w}_k) \right\|^2 \right]. \end{aligned}$$

Where this last line follows from the fact that for any random variable X , $\mathbf{E} \left[\|X - \mathbf{E}[X]\|^2 \right] \leq \mathbf{E} \left[\|X\|^2 \right]$ (i.e. the second moment is an upper bound for the variance). If the component loss functions f_i satisfy $\|\nabla f(x) - \nabla f(y)\| \leq L \cdot \|x - y\|$ for some constant L , then the variance is bounded by

$$\mathbf{E} \left[\left\| (\nabla f_{i_t}(w_{k,t}) - \nabla f_{i_t}(\tilde{w}_k) + \tilde{g}_k) - \nabla f(w_{k,t}) \right\|^2 \right] \leq L^2 \cdot \|w_{k,t} - \tilde{w}_k\|^2$$

Takeaway: the variance of the gradient estimates gets small when the inner iterates $w_{k,t}$ are close to the outer iterates \tilde{w}_k . As SVRG converges (assuming for a moment that the algorithm converges), then both the inner iterates and the outer iterates should approach the stationary point the algorithm is converging to, so this distance should become small. In fact, this can be used to **prove that SVRG converges at a linear rate** (i.e. the same type of rate as gradient descent) under standard assumptions. SVRG is a good choice when (1) we want the low-training-loss solutions that a linear rate gives us quickly, and (2) n is relatively large so gradient descent would be too expensive.

Problem: sometimes we have not only a large number of training examples n , but a large number of parameters d . This can make training difficult, even when we use all the methods in our toolbox so far, because these help us to scale with n (SGD and SVRG) and the condition number κ (acceleration, preconditioning, and adaptive step sizes), but not with the dimension d .

What methods have you used or seen in previous classes for addressing large d ?

Dimensionality Reduction. Idea: transform our original problem in dimension d into one that has a smaller dimension D . Usually this is done by lowering the dimension of the training examples x_i in empirical risk minimization. The benefits of dimensionality reduction include: using less memory to store the training examples x_i and the model w ; requiring less time to compute each iteration of SGD and related algorithms; and (sometimes) improving the condition number of the ERM problem.

Usually, dimensionality reduction is done using *unsupervised learning* (that is, learning without using the labels y_i). There are many ways this can be done, including

- Random projection: choose a D -dimensional linear subspace at random and project onto it.
- Principal Component Analysis: find the D components of the data with the largest variance, keep those, and throw out all the other components.
- Autoencoders: learn a lower-dimension version of the data using a neural network.

Since these methods are covered in depth in **CS5786**, I'm just going to do a quick overview of them here.

Random projection. This is sometimes called the *Johnson-Lindenstrauss transform* or “JL transform” after the following, somewhat surprising lemma.

Lemma 1 (Johnson-Lindenstrauss lemma). *For any points $x_1, \dots, x_n \in \mathbb{R}^d$, any error tolerance $0 < \epsilon < 1$, and any target dimension D that satisfies $D > 8 \log(n)/\epsilon^2$, there is a matrix $A \in \mathbb{R}^{D \times d}$ (i.e. a linear map from \mathbb{R}^d to \mathbb{R}^D) such that for all $i, j \in \{1, \dots, n\}$*

$$(1 - \epsilon) \|x_i - x_j\|^2 \leq \|Ax_i - Ax_j\|^2 \leq (1 + \epsilon) \|x_i - x_j\|^2.$$

What this lemma says is that there is a (linear!) map from \mathbb{R}^d to \mathbb{R}^D that preserves all pairwise distances between the points in the dataset to within a relative error of at most ϵ , as long as D is large enough. But how large D has to be is actually completely independent of d (the dimension of the original points) and only depends *logarithmically* on the number of points in the dataset. So if all we care about is preserving pairwise distances between the points, we can often reduce the dimension substantially, since $\log(n)$ is never very large (e.g. even for a dataset with a million examples it is less than 14).

How do we actually find this map? It turns out that a random map will do the trick! If we pick each

entry of A as an independent Gaussian random variable (with zero mean and appropriate variance) then the bound in the lemma will hold with high probability, as long as the dimension D is large enough.

An aside: concerns with efficiency. While using Gaussian random variables is sufficient, it's not very computationally efficient to generate the matrix A , communicate it, and multiply by it. There's a lot of work into making random projections faster by using other distributions and more structured matrices, so if you want to use random projection at scale, you should consider using these methods.

Principal component analysis (PCA). Idea: instead of using a random linear projection, pick an orthogonal linear map that maximizes the variance of the resulting transformed data. Concretely, if we're given some data $x_1, \dots, x_n \in \mathbb{R}^d$, we want to find an orthonormal matrix $A \in \mathbb{R}^{D \times d}$ (i.e. a matrix with orthogonal rows all of norm 1) that maximizes

$$\frac{1}{n} \sum_{i=1}^n \left\| Ax_i - \frac{1}{n} \sum_{j=1}^n Ax_j \right\|^2 \quad \text{over orthogonal projections } A \in \mathbb{R}^{D \times d}.$$

This problem can be solved by forming the empirical covariance matrix Σ of the data, where

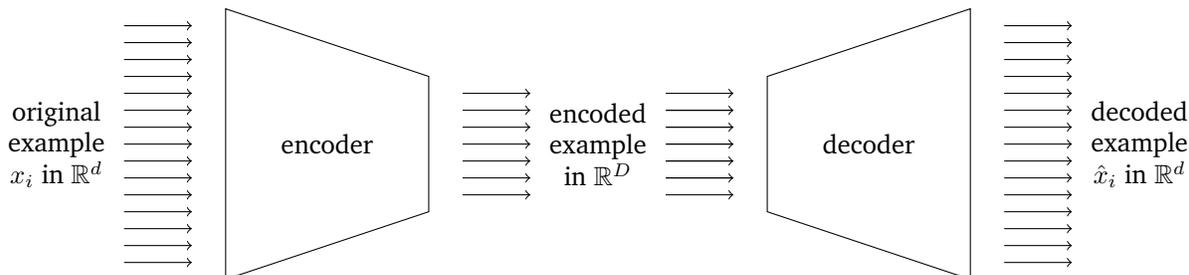
$$\Sigma = \frac{1}{n} \sum_{i=1}^n \left(x_i - \frac{1}{n} \sum_{j=1}^n x_j \right) \left(x_i - \frac{1}{n} \sum_{j=1}^n x_j \right)^T,$$

and then finding its D largest eigenvectors and using them as the rows of A . One downside of this direct approach is that doing so requires $O(d^2)$ space (to store the covariance matrix) and even more time (to do the eigendecomposition). As a result, many methods for fast PCA have been developed, and you should consider using these if you want to use PCA at scale.

Autoencoders. Idea: use deep learning to learn two nonlinear models, one of which (the *encoder*, ϕ) goes from our original data in \mathbb{R}^d to a compressed representation in \mathbb{R}^D for $D < d$, and the other of which (the *decoder*, ψ) goes from the compressed representation in \mathbb{R}^D back to \mathbb{R}^d . We want to train in such a way as to minimize the distance between the original examples in \mathbb{R}^d and the "recovered" examples that result from encoding and then decoding the example. Formally, given some dataset x_1, \dots, x_n , we want to minimize

$$\frac{1}{n} \sum_{i=1}^n \|\psi(\phi(x_i)) - x_i\|^2$$

over some parameterized class of nonlinear transformations $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^D$ and $\psi : \mathbb{R}^D \rightarrow \mathbb{R}^d$ defined by a neural network.



Sparsity. Another way of dealing with large d is to use *sparsity*. A vector or matrix is informally called *sparse* when few of its entries are non-zero. The *density* of a sparse matrix or vector is the fraction of its entries that are non-zero. For example, the vector

$$[3 \ 0 \ 0 \ 0 \ 2 \ 0 \ 0 \ 0 \ 7 \ 0]$$

has density $3/10 = 0.3$. When the density of a matrix is low, we can store and compute with it in a format specialized for sparse matrices. This results in computations that have cost proportional to the number of nonzero entries of the matrix, rather than its dimensions. So if d is large, but the matrices involved have low density, we can often save a lot of compute by using sparse matrix formats.

Storing sparse vectors. The standard way to store a sparse vector is to store only the nonzero entries as pairs consisting of the index and the value of the nonzero entry. Concretely, for the vector above, we would store it as

$$(\text{index: } 0, \text{value: } 3), (\text{index: } 4, \text{value: } 2), (\text{index: } 8, \text{value: } 7).$$

Usually this is stored more compactly as two arrays: one array of indexes and one array of values.

$$\begin{aligned} \text{indexes: } & [0 \ 4 \ 8] \\ \text{values: } & [3 \ 2 \ 7]. \end{aligned}$$

Storing sparse matrices. There are many ways to store a sparse matrix. The simplest way is a direct adaptation of the way to store sparse vectors: we store the indexes (as a row/column pair) and values of all the nonzero entries. This format is called “coordinate list” or **COO**. For example, the matrix

$$\begin{bmatrix} 5 & 0 & 0 & 3 & 0 & 1 \\ 0 & 4 & 0 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 3 & 0 \end{bmatrix}$$

could be encoded in COO as

$$\begin{aligned} \text{row indexes: } & [0 \ 0 \ 0 \ 1 \ 2 \ 2 \ 2] \\ \text{column indexes: } & [0 \ 3 \ 5 \ 1 \ 0 \ 1 \ 4] \\ \text{values: } & [5 \ 3 \ 1 \ 4 \ 1 \ 2 \ 3]. \end{aligned}$$

Note that for COO, the order of the nonzero entries does not matter and no particular order is specified, although they are often sorted for performance reasons. Another common format is “compressed sparse row” or **CSR**. CSR effectively stores all the rows of the matrix as sparse vectors concatenated into one big array, and then uses another row offset array to index into it. The above matrix encoded in CSR would look like

$$\begin{aligned} \text{row offsets: } & [0 \ 3 \ 4] \\ \text{column indexes: } & [0 \ 3 \ 5 \ 1 \ 0 \ 1 \ 4] \\ \text{values: } & [5 \ 3 \ 1 \ 4 \ 1 \ 2 \ 3] \end{aligned}$$

where 0, 3, and 4 are the offsets within the column index and values arrays at which rows 0, 1, and 2 begin, respectively. “Compressed sparse column,” or **CSC** is just the transpose-dual of CSR: it stores the columns of a matrix as sparse vectors rather than the rows.

Comparison of storage formats. While COO is better for building matrices by adding entries, CSR usually allows for faster matrix multiply with dense vectors and matrices.

Other storage formats. There are many other ways to store sparse matrices, especially in the case where the matrix has some structure (for example, if it is banded matrix or a symmetric matrix). Some sparse matrix formats take advantage of dense sub-blocks within the matrix, which they store in a dense format to save memory and compute.