Lecture 11: Dimensionality Reduction and Sparsity.

CS4787 — Principles of Large-Scale Machine Learning Systems

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 $x_1, \ldots, x_n \in \mathbb{R}^d$, 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 $i, j \in \{1, \ldots, n\}$*

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

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 random variable (with zero mean and appropriate variance) then the bound in the lemma will hold with high probability, for large enough D. Here's a sketch the derivation of why we should expect this result to be true.

Suppose that we sample A such that the entries A_{ij} are independent and all distributed according to $\mathcal{N}(0, \sigma^2)$ (where $\mathcal{N}$ denotes the normal distribution) for some constant σ^2 (to be set later). This is basically the simplest zero-mean distribution we can think of. With this, we have that for any point $z \in \mathbb{R}^d$,

$$\begin{aligned}
\mathbf{E}\left[\|Az\|^2\right] &= \mathbf{E}\left[\sum_{i=1}^{D}\left(\sum_{j=1}^{d} A_{ij} z_j\right)^2\right] = \sum_{i=1}^{D} \mathbf{E}\left[\left(\sum_{j=1}^{d} A_{ij} z_j\right)^2\right] \\
&= \sum_{i=1}^{D}\sum_{j=1}^{d} \mathbf{E}\left[(A_{ij} z_j)^2\right] \qquad \text{(because the } A_{ij} \text{ are independent)} \\
&= \sum_{i=1}^{D}\sum_{j=1}^{d} \mathbf{E}\left[A_{ij}^2\right] z_j^2 = \sum_{i=1}^{D}\sum_{j=1}^{d} \sigma^2 z_j^2 = D\sigma^2 \cdot \sum_{j=1}^{d} z_j^2 = D\sigma^2 \cdot \|z\|^2 .
\end{aligned}$$

Now, we choose $\sigma^2 = D^{-1}$, so that these terms cancel and we have

$$\mathbf{E}\left[\|Az\|^2\right] = \|z\|^2 .$$

So we know that $\|Az\|^2$ is an unbiased estimator of $\|z\|^2$, and in particular *distances are preserved in expectation*. Now, if we look a little more closely at the actual distribution of $\|Az\|^2$, we can see that it is given by

$$\|Az\|^2 = \sum_{i=1}^{D} \left(e_i^T Az\right)^2 = \sum_{i=1}^{D} \Big(\underbrace{\sum_{j=1}^{d} A_{ij} z_j}_{\text{Gaussian-distributed}}\Big)^2 .$$

Each of these underbraced terms is just the sum of d Gaussian random variables, which we know is just a Gaussian distribution! We can see that $\sum_{j=1}^{d} A_{ij} z_j \sim \mathcal{N}(0, D^{-1}\|z\|^2)$. And the whole term $\|Az\|^2$ is just the sum of D of these, and they're independent since we sampled the entries of A independently. Since we're summing up a bunch of independent random variables, we should expect (by the central limit theorem) them to concentrate and start behaving like a Gaussian for large enough D.

Now, at this point we'd like to use a concentration inequality (like Hoeffding's), but we're blocked from using Hoeffding's directly because the terms in this sum are not bounded from above. Since we're using Gaussians, we can leverage some special theory about Gaussians to get us the rest of the way instead. Essentially, the trick is to notice that $D \cdot \|z\|^{-2} \cdot \|Az\|^2$ is distributed as the sum of the squares of D independent standard Gaussian random variables, which is exactly a χ^2_D distribution. We can then use a known tail bound on the χ^2_D distribution to prove the lemma. Since this is not particularly instructive (just some algebra), instead I'll **show you that this works with a demo**.

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). We saw that random projection can do a good job of preserving distances, but weirdly, we *didn't use the data* at all to construct the random projection. Can we use the data to somehow "do better" than random projection?

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 $x_1, \ldots, x_n \in \mathbb{R}^d$, 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

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

Note that this objective is equal to

$$\frac{1}{n}\sum_{i=1}^{n}\left(Ax_i - \frac{1}{n}\sum_{j=1}^{n}Ax_j\right)^T\left(Ax_i - \frac{1}{n}\sum_{j=1}^{n}Ax_j\right) = \mathbf{tr}\left(A\left(\frac{1}{n}\sum_{i=1}^{n}\left(x_i - \frac{1}{n}\sum_{j=1}^{n}x_j\right)\left(x_i - \frac{1}{n}\sum_{j=1}^{n}x_j\right)^T\right)A^T\right).$$

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

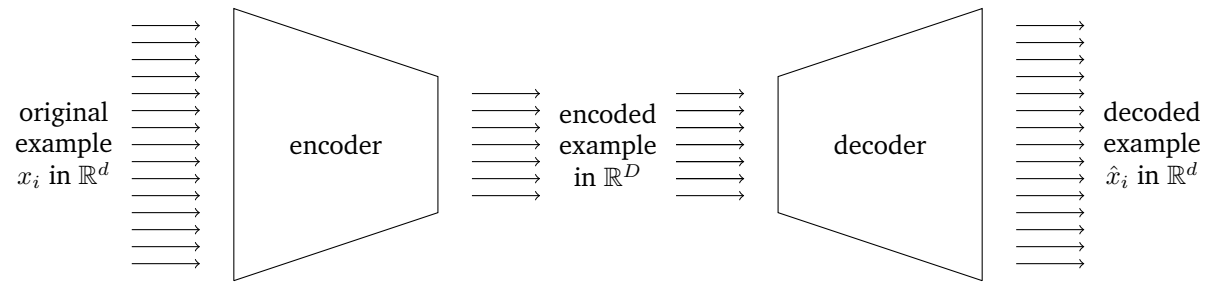
$$\Sigma = \frac{1}{n}\sum_{i=1}^{n}\left(x_i - \frac{1}{n}\sum_{j=1}^{n}x_j\right)\left(x_i - \frac{1}{n}\sum_{j=1}^{n}x_j\right)^T,$$

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 $x_1, \ldots, x_n$, 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 $\phi : \mathbb{R}^d \to \mathbb{R}^D$ and $\psi : \mathbb{R}^D \to \mathbb{R}^d$ 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

$$\begin{bmatrix}3 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 7 & 0\end{bmatrix}$$

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 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

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

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

$$\begin{aligned}\texttt{indexes:} &\quad \begin{bmatrix}0 & 4 & 8\end{bmatrix}\\ \texttt{values:} &\quad \begin{bmatrix}3 & 2 & 7\end{bmatrix}.\end{aligned}$$

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

$$\begin{aligned}\texttt{row indexes:} &\quad \begin{bmatrix}0 & 0 & 0 & 1 & 2 & 2 & 2\end{bmatrix}\\ \texttt{column indexes:} &\quad \begin{bmatrix}0 & 3 & 5 & 1 & 0 & 1 & 4\end{bmatrix}\\ \texttt{values:} &\quad \begin{bmatrix}5 & 3 & 1 & 4 & 1 & 2 & 3\end{bmatrix}.\end{aligned}$$

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

$$\begin{aligned}\texttt{row offsets:} &\quad \begin{bmatrix}0 & 3 & 4\end{bmatrix}\\ \texttt{column indexes:} &\quad \begin{bmatrix}0 & 3 & 5 & 1 & 0 & 1 & 4\end{bmatrix}\\ \texttt{values:} &\quad \begin{bmatrix}5 & 3 & 1 & 4 & 1 & 2 & 3\end{bmatrix}\end{aligned}$$

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.