CPSC 540: Machine Learning

Kernel Methods

Mark Schmidt

University of British Columbia

Winter 2018

Last time: Stochastic Average Gradient (SAG)

- We discussed stochastic gradient methods minimizing finite sums,

$$f(w) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(w),$$

- For Lipschitz ∇f and strongly-convex f, ways to get linear convergence:
  - Grow the batch size $|\mathcal{B}^k|$ fast enough,

    $$w^{k+1} = w^k - \frac{\alpha_k}{|\mathcal{B}^k|}\sum_{i \in \mathcal{B}^k} f_i(w^k),$$

    makes setting step-size easy but eventually needs all gradients on each iteration.
  - Stochastic average gradient (SAG),

    $$w^{k+1} = w^k - \frac{\alpha_k}{n}\sum_{i=1}^{n} v_i^k,$$

    where on each step we set $v_{i_k}^k = \nabla f_{i_k}(w^k)$ for one random i_k.
    - Only evaluates one gradient per iteration.

Stochastic Average Gradient

- We can think of SAG as having a memory:

$$\begin{bmatrix} \text{———} & v_1 & \text{———} \\ \text{———} & v_2 & \text{———} \\ & \vdots & \\ \text{———} & v_n & \text{———} \end{bmatrix},$$

  where v_i^k is the gradient $\nabla f_i(w^k)$ from the last k where i was selected.

- On each iteration we:
  - Randomly choose one of the v_i and update it to the current gradient.
  - We take a step in the direction of the avarge of these v_i.

SAG Algorithm

- Basic SAG algorithm (maintains $g = \sum_{i=1}^{n} v_i$):
    - Set $g = 0$ and gradient approximation $v_i = 0$ for $i = 1, 2, \dots, n$.
    - while(1)
        - Sample i from $\{1, 2, \dots, n\}$.
        - Compute $\nabla f_i(w)$.
        - $g = g - v_i + \nabla f_i(w)$.
        - $v_i = \nabla f_i(w)$.
        - $w = w - \frac{\alpha}{n} g$.
- Iteration cost is $O(d)$, but "lazy updates" allow $O(z)$ with sparse gradients.
- For linear models where $f_i(w) = h(w^T x^i)$, it only requires $O(n)$ memory:

$$\nabla f_i(w) = \underbrace{h'(w^T x^i)}_{\text{scalar}} \underbrace{x^i}_{\text{data}}.$$

    - Least squares, logistic regression, etc.
- For neural networks, would need to store all activations (which seems bad).

Discussion of SAG and Beyond

- Bonus slides discuss practical issues related to SAG:
  - Setting step-size with an approximation to L.
  - Deciding when to stop.
  - Lipschitz sampling of training examples.
    - Improves rate for SAG, only changes constants for SG.
- There are now a bunch of stochastic algorithm with $O(\log(1/\epsilon))$ rates:
  - SDCA, MISO, mixedGrad, SVRG, S2GD, Finito, SAGA, etc.
  - Accelerated/Newton-like/coordinate-wise/proximal/ADMM versions.
  - Analysis in non-convex settings, including new algorithms for PCA.
- Most notable is SVRG which gets rid of the memory...

Stochastic Variance-Reduced Gradient (SVRG)

SVRG algorithm: gets rid of memory by occasionally computing exact gradient.

- Start with w_0
- for $s = 0, 1, 2 \ldots$
    - $\nabla f(w_s) = \frac{1}{n}\sum_{i=1}^n \nabla f_i(w_s)$
    - $w^0 = w_s$
    - for $k = 0, 1, 2, \ldots m$
        - Randomly pick $i_k \in \{1, 2, \ldots, n\}$
        - $w^{k+1} = w^k - \alpha_k(\nabla f_{i_k}(w^k) - \underbrace{\nabla f_{i_k}(w_s) + \nabla f(w_s)}_{\text{mean zero}})$.
    - $w_{s+1} = w^k$.

Convergence properties similar to SAG (for suitable m).

- Unbiased: $\mathbb{E}[\nabla f_{i_k}(w_s)] = \nabla f(w_s)$ (special case of "control variate").
- Theoretically m depends on L, μ, and n.
- In practice $m = n$ usually works.
    - $O(d)$ storage at average cost of 3 gradients per iteration.

Stochastic Subgradient for Infinite Datasets?

- Our analysis of stochastic subgradient used two assumptions on g_{i_t}:
  - Unbiased approximation of subgradient: $\mathbb{E}[g_{i_t}] = g_t$.
  - Variance is bounded: $\mathbb{E}[\|g_{i_t}\|^2] \leq B^2$.

- Consider a scenario where we have infinite number of IID samples:
  - We can optimize the test loss,

$$\operatorname*{argmin}_{w \in \mathbb{R}^d} \mathbb{E}[f_i(w)],$$

    by applying stochastic subgradient on a new IID sample on each iteration.
  - In this setting, we are directly optimizing test loss and cannot overfit.
  - We require $O(1/\epsilon)$ samples to reach test loss accuracy of ϵ (under PL).

- However, keep in mind that the test loss may not be the test error.
  - Linear classifiers approximate 0-1 loss (test error) with logistic/hinge loss (test loss).

Infinite-Data Optimization

- Consider number of training examples so large we can't go through all examples.
    - Stochastic gradient gets within ϵ of optimal test loss after $t = O(1/\epsilon)$ iterations.
- How does this compare to sampling t examples and optimizing on these?
    - What we usually do: "minimize regularized training loss".
- How many samples t before training objective is within ϵ of test objective?
    - Minimum possible assumptions: $t = O(1/\epsilon^2)$.
    - Realistic assumptions: $t = O(1/\epsilon)$.
    - Strong assumptions: $t = O(\log(1/\epsilon))$.
- "Realistic": n iterations of stochastic gradient on n examples is optimal!?!
    - Almost always worse empirically than methods which do multiple passes.
    - Constants matter for test data (better optimization improves constants).

End of Part 1: Key Ideas

- Typical ML problems are written as optimization problem

$$\underset{w\in\mathbb{R}^d}{\text{argmin}}\, F(w) = \frac{1}{n}\sum_{i=1}^{n} f_i(w^T x^i) + \lambda r(w).$$

- Convex optimization packages:
  - For the special case when F is convex and d is small.
- Gradient descent:
  - Applies when F is differentiable, yields iteration cost that is linear in d.
  - Only needs $O(\log(1/\epsilon))$ iterations if F is strongly-convex.
  - Faster versions like Nesterov's and Newton-like methods exist.
- Proximal gradient:
  - Applies when f_i is differentiable and r is "simple" (like L1-regularization).
  - Similar convergence properties to gradient descent, even for non-smooth r.
  - Special case is projected gradient, which allows "simple" constraints.

End of Part 1: Key Ideas

- Typical ML problems are written as optimization problem
$$\operatorname*{argmin}_{w\in\mathbb{R}^d} F(w) = \frac{1}{n}\sum_{i=1}^{n} f_i(w^Tx^i) + \lambda r(w).$$
- Coordinate optimization:
  - Faster than gradient descent if iterations are d-times cheaper.
  - Allows non-smooth r if it's separable.
- Stochastic subgradient:
  - Iteration cost is n-times cheaper than [sub]gradient descent, and allow $n = \infty$.
  - For non-smooth problems, convergence rate is same as subgradient method.
  - For smooth problems, number of iterations is much higher than gradient descent.
- SAG and SVRG:
  - Special case when F is smooth.
  - Same low cost as stochastic gradient methods.
  - But similar convergence rate to gradient descent.

Other Non-Smooth Optimization

- We discussed structured regularization to enforce patterns in w:
  - Total-variation regularizaiton and structured sparsity.
- We can use proximal-gradient versions of the large-scale methods:
  - Coordinate optimization, stochastic subgradient, SAG, and SVRG.
- Keywords for ther common non-smooth methods:
  - Proximal-Newton, Chambolle-Pock, ADMM, Frank-Wolfe, mirror descent.
- In previous years we also covered dual methods:
  - For cases with non-smooth convex f_i and L2-regularization.
  - Transforms into a smooth problem where we can apply coordinate optimization.
  - Similar cost to stochastic subgradient, but you can use line-search to set step-size.
  - If you're interested, I put the slides from last year here: `https://www.cs.ubc.ca/~schmidtm/Courses/540-W18/L12.5.pdf`

Even Bigger Problems?

- What about datasets that don't fit on one machine?
  - We need to consider parallel and distributed optimization.
- New issues:
  - Synchronization: we may not want to wait for the slowest machine.
  - Communication: it's expensive to transfer data and parameters across machines.
  - Failures: in huge-scale settings, machine failure probability is non-trivial.
- "Embarassingly" parallel solution:
  - Split data across machines, each machine computes gradient of their subset.
  - Papers present more fancy methods, but always try this first ("linear speedup").
- Fancier methods:
  - Asyncronous stochastic subgradient (works fine if you make the step-size smaller).
  - Parallel coordinate optimization (works fine if you make the step-size smaller).
  - Decentralized gradient (needs a smaller step-size and an "EXTRA" trick).

Machine Learning Reading Group

- The machine learning reading group (MLRG) this term:
  - Tuesdays from 5-6 in ICICS 146, starting tomorrow.
- We'll be focusing on parallel and distributed methods this term.

Outline

Motivation: Multi-Dimensional Polynomial Basis

- Consider quadratic polynomial basis with only have two features ($x^i \in \mathbb{R}^2$):

$$\hat{y}^i = w_0 + w_1 x_1^i + w_2 x_2^i + w_2 (x_1^i)^2 + w_3 (x_2^i)^2 + w_4 x_1^i x_2^i.$$

- In 340 we saw that we can fit this model using a change of basis:

$$X = \begin{bmatrix} 0.2 & 0.3 \\ 1 & 0.5 \\ -0.5 & -0.1 \end{bmatrix} \Rightarrow Z = \begin{bmatrix} 1 & 0.2 & 0.3 & (0.2)^2 & (0.3)^2 & 0.2 \cdot 0.3 \\ 1 & 1 & 0.5 & (1)^2 & (0.5)^2 & 1 \cdot 0.5 \\ 1 & -0.5 & -0.1 & (-0.5)^2 & (-0.1)^2 & -0.5 \cdot -0.1 \end{bmatrix}$$

- If you have $d = 100$ and $p = 5$, there are $O(100^5)$ possible degree-5 terms:

$$(x_1^i)^5, (x_1^i)^4 x_2^i, (x_1)^4 x_3^i, \dots, (x_1^i)^3 (x_2^i)^2, (x_1^i)^3 (x_2^i)^2, \dots, (x_1^i)^3 x_2^i x_3^i, \dots$$

- How can we do this when number of features k in basis is huge?

The "Other" Normal Equations

- Recall the L2-regularized least squares model with basis Z,

$$\underset{v \in \mathbb{R}^d}{\text{argmin}} \frac{1}{2}\|Zv - y\|^2 + \frac{\lambda}{2}\|v\|^2.$$

- By solving for $\nabla f(v) = 0$ we get that

$$v = (\underbrace{Z^TZ}_{k \text{ by } k} + \lambda I_d)^{-1} Z^T y,$$

  where I_d is the d by d identity matrix.
- An equivalent way to write the solution is:

$$v = Z^T(\underbrace{ZZ^T}_{n \text{ by } n} + \lambda I_n)^{-1} y,$$

  by using a variant of the matrix inversion lemma (bonus slide).
- Computing v with this formula is faster if $n << d$:
  - ZZ^T is n by n while Z^TZ is d by d.

Predictions using Equivalent Form

- Given test data $\tilde{X}$, we predict $\hat{y}$ using:

$$\begin{aligned}\hat{y} &= \tilde{Z}v \\ &= \tilde{Z}\underbrace{Z^T(ZZ^T + \lambda I_n)^{-1}y}_{\text{“other” normal equations}}\end{aligned}$$

- If we define $K = ZZ^T$ (Gram matrix) and $\tilde{K} = \tilde{Z}Z^T$, then we have

$$\hat{y} = \tilde{K}(K + \lambda I_n)^{-1}y,$$

  where K is $n \times n$ and $\tilde{K}$ is $t \times n$.

- Key observation behind kernel trick:
  - If we can directly compute K and $\tilde{K}$, we don't need to form Z or $\tilde{Z}$.

Gram Matrix

- The Gram matrix K is defined by:

$$K = ZZ^T = \begin{bmatrix} — & (z^1)^T & — \\ — & (z^2)^T & — \\ & \vdots & \\ — & (z^n)^T & — \end{bmatrix} \begin{bmatrix} | & | & | & \cdots & | \\ z^1 & z^2 & z^3 & \cdots & z^n \\ | & | & | & \cdots & | \end{bmatrix}$$

$$= \begin{bmatrix} \langle z^1, z^1 \rangle & \langle z^1, z^2 \rangle & \cdots & \langle z^1, z^n \rangle \\ \langle z^2, z^1 \rangle & \langle z^2, z^2 \rangle & \cdots & \langle z^2, z^n \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle z^n, z^1 \rangle & \langle z^n, z^2 \rangle & \cdots & \langle z^n, z^n \rangle \end{bmatrix} = \begin{bmatrix} k(x^1, x^1) & k(x^1, x^2) & \cdots & k(x^1, x^n) \\ k(x^2, x^1) & k(x^2, x^2) & \cdots & k(x^2, x^n) \\ \vdots & \vdots & \ddots & \vdots \\ k(x^n, x^1) & k(x^n, x^2) & \cdots & k(x^n, x^n) \end{bmatrix}$$

- K contains the inner products between all training examples in basis z
- $\tilde{K}$ contains the inner products between training and test examples.
  - Kernel trick: if we can compute $k(x^i, x^j) = \langle z^i, z^j \rangle$, we don't need z^i and z^j.

Polynomial Kernel

- In 340 we saw the polynomial kernel of degree p,

$$k(x^i, x^k) = (1 + \langle x^i, x^j \rangle)^p,$$

  which corresponds to a general degree-p polynomial z^i.

- You can make predictions with these z^i using

$$\hat{y} = \tilde{K}(K + \lambda I)^{-1} y.$$

- Total cost is only $O(n^2 d + n^3)$ even though number of features is $O(d^p)$.

- Kernel trick:
  - We have kernel function $k(x^i, x^j)$ that gives $\langle z^i, z^j \rangle$.
  - Skip forming Z and directly form K and $\tilde{K}$.
  - Size of K is n by n even if Z has exponential or infinite columns.

Guasian-RBF Kernels

- The most common kernel is the Gaussian-RBF (or 'squared exponential') kernel,

$$k(x^i, x^j) = \exp\left(-\frac{\|x^i - x^j\|^2}{2\sigma^2}\right).$$

- What features z_i would lead to this as the inner-product?
  - To simplify, assume $d = 1$ and $\sigma = 1$,

$$k(x^i, x^j) = \exp\left(-\frac{1}{2}(x^i)^2 + x^i x^j - \frac{1}{2}(x^j)^2\right) = \exp\left(-\frac{1}{2}(x^i)^2\right)\exp(x^i x^j)\exp\left(-\frac{1}{2}(x^j)^2\right),$$

  so we need $z_i = \exp(-\frac{1}{2}(x^i)^2)u_i$ where $u_i u_j = \exp(x^i x^j)$.
    - For this to work for *all* x^i and x^j, z_i must be infinite-dimensional.
  - If we use that

$$\exp(x^i x^j) = \sum_{k=0}^{\infty} \frac{(x^i)^k (x^j)^k}{k!},$$

  then we obtain

$$z_i = \exp\left(-\frac{1}{2}(x^i)^2\right)\left[1 \quad \frac{1}{\sqrt{1!}}x^i \quad \frac{1}{\sqrt{2!}}(x^i)^2 \quad \frac{1}{\sqrt{3!}}(x^i)^3 \quad \cdots\right].$$

Kernel Trick for Structured Data

- Kernel trick can be useful for structured data:
  - Consider that doesn't look like this:

$$X = \begin{bmatrix} 0.5377 & 0.3188 & 3.5784 \\ 1.8339 & -1.3077 & 2.7694 \\ -2.2588 & -0.4336 & -1.3499 \\ 0.8622 & 0.3426 & 3.0349 \end{bmatrix}, \quad y = \begin{bmatrix} +1 \\ -1 \\ -1 \\ +1 \end{bmatrix},$$

but instead looks like this:

$$X = \begin{bmatrix} \text{Do you want to go for a drink sometime?} \\ \text{J'achète du pain tous les jours.} \\ \text{Fais ce que tu veux.} \\ \text{There are inner products between sentences?} \end{bmatrix}, \quad y = \begin{bmatrix} +1 \\ -1 \\ -1 \\ +1 \end{bmatrix}.$$

  - It might be easier to define a "similarity" between sentences than to define features.

Kernel Trick for Structured Data

- A classic "string kernel":
  - We want to compute k("cat", "cart").
  - Find common subsequences: 'c', 'a', 't', 'ca', 'at', 'ct', 'cat'.
  - Weight them by total length in original strings:
    - 'c' is has lengths (1,1), 'ca' has lengths (2,2), 'ct' has lengths (3,4), and son.
  - Add up the weighted lengths of common subsequences to get a similarity:

$$\text{k(“cat”,“cart')} = \underbrace{\gamma^1\gamma^1}_{\text{‘c'}} + \underbrace{\gamma^1\gamma^1}_{\text{‘a'}} + \underbrace{\gamma^1\gamma^1}_{\text{‘t'}} + \underbrace{\gamma^2\gamma^2}_{\text{‘ca'}} + \underbrace{\gamma^2\gamma^3}_{\text{‘at'}} + \underbrace{\gamma^3\gamma^4}_{\text{‘ct'}} + \underbrace{\gamma^3\gamma^4}_{\text{‘cat'}},$$

    where γ is a hyper-parameter controlling influence of length.
  - Corresponds to exponential feature set (counts/lengths of all subsequences).
    - But kernel can be computed in linear time by dynamic programming.

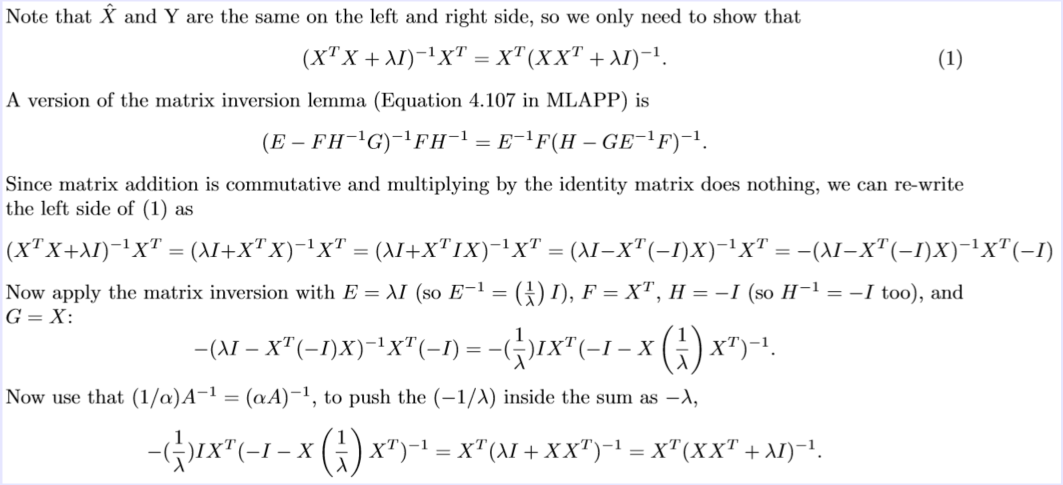
- Many variations exist. And there are "image kernels", "graph kernels", and so on.
  - You can turn probabilities over examples (second half of course) into kernels.
  - A survey on the topic is here.

Summary

- SVRG removes the memory requirement of SAG.
- Infinite datasets can be handle with stochastic subgradient methods.
    - This is theoretically "optimal" in some settings, not optimal in practice.
- Kernel trick: allows working with "similarity" instead of features.
    - Also allows exponential- or infinite-sized feature spaces.

- Next time:
    - Instead of predicting scalar label y^i, we want to predict sentences/images/proteins.

Equivalent Form of Ridge Regression



Note that $\hat{X}$ and Y are the same on the left and right side, so we only need to show that

$$(X^T X + \lambda I)^{-1} X^T = X^T (X X^T + \lambda I)^{-1}. \tag{1}$$

A version of the matrix inversion lemma (Equation 4.107 in MLAPP) is

$$(E - FH^{-1}G)^{-1} FH^{-1} = E^{-1} F (H - GE^{-1}F)^{-1}.$$

Since matrix addition is commutative and multiplying by the identity matrix does nothing, we can re-write the left side of (1) as

$$(X^T X + \lambda I)^{-1} X^T = (\lambda I + X^T X)^{-1} X^T = (\lambda I + X^T I X)^{-1} X^T = (\lambda I - X^T(-I)X)^{-1} X^T = -(\lambda I - X^T(-I)X)^{-1} X^T(-I)$$

Now apply the matrix inversion with $E = \lambda I$ (so $E^{-1} = \left(\frac{1}{\lambda}\right) I$), $F = X^T$, $H = -I$ (so $H^{-1} = -I$ too), and $G = X$:

$$-(\lambda I - X^T(-I)X)^{-1} X^T(-I) = -(\frac{1}{\lambda}) I X^T (-I - X \left(\frac{1}{\lambda}\right) X^T)^{-1}.$$

Now use that $(1/\alpha)A^{-1} = (\alpha A)^{-1}$, to push the $(-1/\lambda)$ inside the sum as $-\lambda$,

$$-(\frac{1}{\lambda}) I X^T (-I - X \left(\frac{1}{\lambda}\right) X^T)^{-1} = X^T (\lambda I + X X^T)^{-1} = X^T (X X^T + \lambda I)^{-1}.$$

SAG Practical Implementation Issues

- Implementation tricks:
  - Improve performance at start using $\frac{1}{m}g$ instead of $\frac{1}{n}g$.
    - m is the number of examples visited.
  - Common to use $\alpha_k = 1/L$ and use adaptive L.
    - Start with $\hat{L} = 1$ and double it whenever we don't satisfiy

$$f_{i_k}\left(w^k - \frac{1}{\hat{L}}\nabla f_{i_k}(w^k)\right) \leq f_{i_k}(w^k) - \frac{1}{2\hat{L}}\|\nabla f_{i_k}(w^k)\|^2,$$

and $\|\nabla f_{i_k}(w^k)\|$ is non-trivial. Costs $O(1)$ for linear models in terms of n and d.

  - Can use $\|w^{k+1} - w^k\|/\alpha = \frac{1}{n}\|g\| \approx \|\nabla f(w^k)\|$ to decide when to stop.
  - Lipschitz sampling of examples improves convergence rate:
    - As with coordinate descent, sample the ones that can change quickly more often.
    - For classic SG methods, this only changes constants.