CPSC 540: Machine Learning

Stochastic Average Gradient, Kernel Methods

Mark Schmidt

University of British Columbia

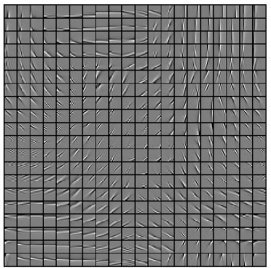
Winter 2017

Admin

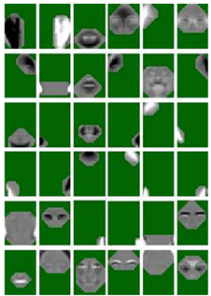
- Assignment 2:
  - Due February 6 (1.5 weeks).
  - Start early, use Piazza.
- Office hours time/location change for this week only:
  - 2:15-3 on Friday in ICICS 193.

Last Time: Structured Sparsity

- Beyond sparsity, we can use regularization to encourage other patterns:
  - Total-variation regularization encourages slow/sparse changes in w.
  - Nuclear-norm regularization encourages sparsity in rank of matrices.
  - Overlapping group L1-regularization encourages sparsity in variable patterns.



$$W = \begin{bmatrix} | & | \\ u_1 & u_2 \\ | & | \end{bmatrix} \begin{bmatrix} | & | \\ v_1 & v_2 \\ | & | \end{bmatrix}^T$$



http://lear.inrialpes.fr/people/mairal/resources/pdf/review_sparse_arxiv.pdf

https://arxiv.org/pdf/1109.2397v2.pdf

- These regularizers are not simple, but solvers are available:
  - Inexact proximal-gradient, ADMM, Frank-Wolfe, UV^T parameterization.

Last time: Stochastic sub-gradient

- We discussed minimizing finite sums,

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

  when n is very large.

- For non-smooth f_i, we discussed stochastic subgradient method,

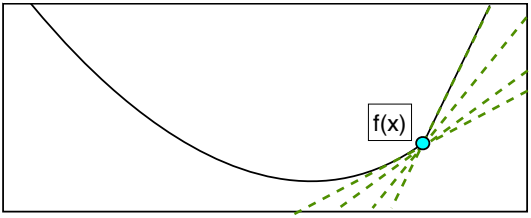
$$x^{t+1} = x^t - \alpha g_{i_t},$$

  for some $g_{i_t} \in \partial f_{i_t}(x^t)$ for some random $i_t \in \{1, 2, \dots, n\}$.
  - May increase f, but moves closer to x^* for small α_t in expectation.

- Same $O(1/\epsilon)$ rate as deterministic subgradient method but n times faster.

Last Time: Subgradients and Subgradient Method

- Subgradients are a generalization of gradients for non-smooth optimization.
  - Slopes of linear underestimators, set of subgradients at x is sub-differential $\partial f(x)$.



  - If at a differentiable x, gradient is the only subgradient.
  - Subgradients exist everywhere for convex funcitons (except vertical asymptotes).
  - We can define them locally for non-convex functions.
    - Called "Clarke" or "Frechet" subgradients.

Last Time: Calculating Subgradients

- Computing general subgradient is complicated, but if f_1 and f_2 are convex then

$$d \in \partial(f_1(x) + f_2(x)) \text{ if } d = d_1 + d_2 \text{ for } d_1 \in \partial f_1(x) \text{ and } d_2 \in \partial f_2(x).$$

$$\partial \max\{f_1(x), f_2(x)\} = \begin{cases} \nabla f_1(x) & f_1(x) > f_2(x) \\ \nabla f_2(x) & f_2(x) > f_1(x) \\ \theta \nabla f_1(x) + (1-\theta)\nabla f_2(x) & f_1(x) = f_2(x) \end{cases}$$

- So for SVMs,

$$f(w) = \frac{1}{n}\sum_{i=1}^{n} \max\{0, 1 - y^i(w^T x^i)\} + \frac{\lambda}{2}\|w\|^2,$$

we can get a sub-gradient by computing

$$\frac{1}{n}\sum_{i=1}^{n} g_i + \lambda w, \text{ with } g_i = \begin{cases} -y^i x^i & \text{if } 1 - y^i(w^T x^i) > 0 \\ 0 & \text{otherwise} \end{cases}$$

What is the best subgradient?

- We considered the deterministic subgradient method,

$$x^{t+1} = x^t - \alpha_t g_t, \text{ where } g_t \in \partial f(x^t),$$

  under any choice of subgradient.

- But what is the "best" subgradient to use?
  - Convex functions have directional derivatives everywhere.
  - Direction $-g_t$ that minimizes directional derivative is minimum-norm subgradient,

$$g^t = \underset{g \in \partial f(x^t)}{\text{argmin}} \; ||g||$$

  - This is the steepest descent direction for non-smooth convex optimization problems.

  - You can compute this for L1-regularization, but not many other problems.
  - Used in best deterministic L1-regularization methods, combined with Newton.

Outline

Stochastic Subgradient with Sparse Features

- We've discussed the number of iterations.
- But a new issue arises regarding the iteration cost.
  - In high-level languages like Matlab, stochastic subgradient might be slow.
  - We need to deal with sparsity of features.

- For many datasets, our feature vectors x^i are very sparse:

"CPSC	"Expedia"	"vicodin"	<recipient name>	...
1	0	0	0	...
0	1	0	0	...
0	0	1	0	...
0	1	0	1	...
1	0	1	1	...

- Consider case where d is huge but each row x^i has at most k non-zeroes:
  - The $O(d)$ cost of stochastic subgradient might be too high.
  - We can often modify stochastic subgradient to have $O(k)$ cost.

Digression: Operations on Sparse Vectors

- Consider a vector $g \in \mathbb{R}^d$ with at most k non-zeroes:

$$g^T = \begin{bmatrix} 0 & 0 & 0 & 1 & 2 & 0 & -0.5 & 0 & 0 & 0 \end{bmatrix}.$$

- If $k << d$, we can store the vector using $O(k)$ storage instead of $O(d)$:
  - Just store the non-zero values:

$$g_{\text{value}}^T = \begin{bmatrix} 1 & 2 & -0.5 \end{bmatrix}.$$

  - Store index of each non-zero ("pointer"):

$$g_{\text{point}}^T = \begin{bmatrix} 4 & 5 & 7 \end{bmatrix}.$$

- With this representation, we can do standard vector operations in $O(k)$:
  - Compute αg in $O(k)$ by computing αg_{value}.
  - Compute $(w - g)$ in $O(k)$ for dense w by subracting g_{value} from w at positions g_{point}.

Stochastic Subgradient with Sparse Features

- Consider optimizing the hinge-loss,

$$\underset{w\in\mathbb{R}^d}{\text{argmin}} \frac{1}{n}\sum_{i=1}^{n}\max\{0, 1-y^i(w^Tx^i)\},$$

when d is huge but each x^i has at most k non-zeroes.

- A stochastic subgradient method could use

$$w^{t+1} = w^t - \alpha_t g_{i_t}, \text{ where } g_i = \begin{cases} -y^i x^i & \text{if } 1-y^i(w^Tx^i) > 0 \\ 0 & \text{otherwise} \end{cases}$$

- Calculating w^{t+1} is $O(k)$ since these are sparse vector operations.

- So stochastic subgradient is fast if k is small even if d is large.
  - This is how you "train on all e-mails": each e-mail has a limited number of words.

Stochastic Subgradient with Sparse Features

- But consider the L2-regularized hinge-loss in the same setting,

$$\operatorname*{argmin}_{w\in\mathbb{R}^d} \frac{1}{n}\sum_{i=1}^{n} \max\{0, 1 - y_i(w^T x^i)\} + \frac{\lambda}{2}\|w\|^2,$$

  using a stochastic subgradient method,

$$w^{t+1} = w^t - \alpha_t g_{i_t} - \alpha_t \lambda w^t, \text{ where } g_{i_t} \text{ is same as before}$$

- Problems is that w^t could have d non-zeroes:
  - So adding L2-regularization increases cost from $O(k)$ to $O(d)$?
- To use L2-regularization and keep $O(k)$ cost, re-write iteration as

$$\begin{aligned} w^{t+1} &= w^t - \alpha_t g_{i_t} - \alpha_t \lambda w^t \\ &= \underbrace{(1 - \alpha_t \lambda) w^t}_{\text{changes scale of } w^t} - \underbrace{\alpha_t g_{i_t}}_{\text{sparse update}}. \end{aligned}$$

Stochastic Subgradient with Sparse Features

- Let's write the update as two steps

$$w^{t+\frac{1}{2}} = (1 - \alpha_t\lambda)w^t, \quad w^{t+1} = w^{t+\frac{1}{2}} - \alpha_t g_{i_t}.$$

- We can implement both steps in $O(k)$ if we re-parameterize as

$$w^t = \beta^t v^t,$$

for some scalar β^t and vector v^t.

- For the first step we can use

$$\beta^{t+\frac{1}{2}} = (1 - \alpha_t\lambda)\beta^t, \quad v^{t+\frac{1}{2}} = v^t.$$

which costs $O(1)$.

- For the second step we can use

$$\beta^{t+1} = \beta^{t+\frac{1}{2}}, \quad v^{t+1} = v^{t+\frac{1}{2}} - \frac{\alpha_t}{\beta^{t+\frac{1}{2}}} g_{i_t},$$

which costs $O(k)$.

Stochastic Subgradient with Sparse Features

- There exists efficient sparse updates in other scenarios too:

- "Lazy updates" track cumulative effects of simple updates:
  - Soft-threshold operator with constant step-size α applies to each element,

$$w_j^{t+1} = \text{sign}(w_j^t)\max\{0, |w_j^t| - \alpha\lambda\}.$$

  - If all that happens to w_j for 10 iterations is the proximal operator, we can use

$$w_j^{t+10} = \text{sign}(w_j^t)\max\{0, |w_j^t| - 10\alpha\lambda\}.$$

Stochastic Subgradient Methods in Practice

- Last time we argued that α_t must go to zero for convergence.
- Theory says using $\alpha_t = 1/\mu t$ is close to optimal.
  - Except for some special cases, you should not do this.
  - Usually $\mu = O(1/n)$ or $O(1/\sqrt{n})$ so initial steps are huge.
  - Later steps are tiny: $1/t$ gets small very quickly.
  - Convergence rate slows dramatically if μ isn't accurate.
  - No adaptation to "easier" problems than worst case.
- Decreasing step-sizes are also hard to tune.
- They also make hard to decide when to stop.

Stochastic Subgradient Methods in Practice

- Tricks that can improve theoretical and practical properties:
  1. Use smaller initial step-sizes, that go to zero more slowly:

$$\alpha_t = \gamma/\sqrt{t},$$

     or just use a constant step-size,

$$\alpha_t = \gamma,$$

     which we showed converges linearly to $O(\gamma)$-ball around the solution.
  2. Take a (weighted) average of the iterations or gradients:

$$\bar{x}^t = \sum_{k=1}^{t} \omega_k x^k,$$

     where ω_t is weight at iteration t.
     - Could weight all iterations equally.
     - Could ignore first half of the iterations then weight equally.
     - Could weight proportional to t.

Speeding up Stochastic Subgradient Methods

- Results that support using large steps and averaging:
  - Averaging later iterations achieves $O(1/t)$ in non-smooth case.
  - Gradient averaging improves constants in analysis.
  - $\alpha_t = O(1/t^\beta)$ for $\beta \in (0.5, 1)$ more robust than $\alpha_t = O(1/t)$.
  - Constant step size ($\alpha_t = \alpha$) achieves linear rate to accuracy $O(\alpha)$.
  - In smooth case, iterate averaging is asymptotically optimal:
    - Achieves same rate as optimal stochastic Newton method.

- These tricks usually help, but tuning is often required:
  - Stochastic subgradient is not a black box.

Stochastic Newton Methods?

- Should we use Nesterov/Newton-like stochastic methods?
  - These do not improve the $O(1/\epsilon)$ convergence rate.

- But some positive results exist.
  - Nesterov/Newton improve performance at start or if variance is small.
  - Two-phase Newton-like method achieves $O(1/\epsilon)$ without strong-convexity.
  - AdaGrad method,

$$x^{t+1} = x^t + \alpha D \nabla f_{i_t}(x^t), \quad \text{with } D_{jj} = \sqrt{\sum_{k=1}^{t} \|\nabla_j f_{i_k}(x^t)\|^2},$$

improves "regret" but not optimization error.
    - Popular variations are RMSprop and Adam.

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: $\mathbb{E}[\|g_{i_t}\|^2] \leq B^2$.
- We can achieve this in the general setting:

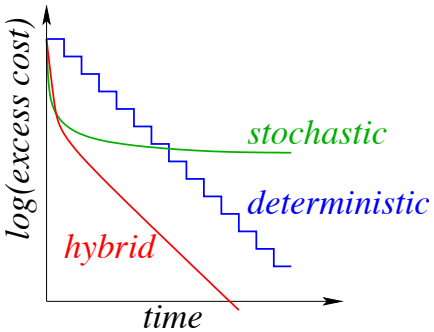
$$\operatorname*{argmin}_{x \in \mathbb{R}^d} \mathbb{E}[f_i(x)].$$

- We can use stochastic subgradient on IID samples from infinite dataset:
  - 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 (optimal).

- Often used to justify doing one "pass" through data of stochastic subgradient:
  - If you only look at data point once, can be viewed as IID test sample.
  - Almost always worse empirically than methods which do multiple passes.

Outline

Better Methods for Smooth Objectives and Finite Datasets?



- Stochastic methods:
  - $O(1/\epsilon)$ iterations but requires 1 gradient per iterations.
  - Rates are unimprovable for general stochastic objectives.
- Deterministic methods:
  - $O(\log(1/\epsilon))$ iterations but requires n gradients per iteration.
  - The faster rate is possible because n is finite.
- For finite n, can we design a better method?

Hybrid Deterministic-Stochastic

- Approach 1: control the sample size.
- Deterministic method uses all n gradients,

$$\nabla f(x^t) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(x^t).$$

- Stochastic method approximates it with 1 sample,

$$\nabla f_{i_t}(x^t) \approx \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(x^t).$$

- A common variant is to use larger sample $\mathcal{B}^t$

$$\frac{1}{|\mathcal{B}^t|}\sum_{i\in\mathcal{B}^t} \nabla f_i(x^t) \approx \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(x^t),$$

particularly useful for vectorization/parallelization.

- For example, with 16 cores set $|\mathcal{B}^t| = 16$ and compute 16 gradients at once.

Approach 1: Batching

- The SG method with a sample $\mathcal{B}^t$ uses iterations

$$x^{t+1} = x^t - \frac{\alpha_t}{|\mathcal{B}^t|} \sum_{i \in \mathcal{B}^t} f_i(x^t).$$

- Let's view this as a "gradient method with error",

$$x^{t+1} = x^t - \alpha_t(\nabla f(x^t) + e^t),$$

where e^t is the difference between approximate and true gradient.

- The batch size $|\mathcal{B}^t|$ controls size of error e^t.
  - If we sample with replacement we get

$$\mathbb{E}[\|e^t\|^2] = \frac{1}{|\mathcal{B}^t|}\sigma^2,$$

where σ^2 is the variance of the gradient norms.
  - Doubling the batch size cuts radius of $O(\alpha)$ ball in half.

Approach 1: Batching

- The SG method with a sample $\mathcal{B}^t$ uses iterations

$$x^{t+1} = x^t - \frac{\alpha_t}{|\mathcal{B}^t|} \sum_{i \in \mathcal{B}^t} f_i(x^t).$$

- Let's view this as a "gradient method with error",

$$x^{t+1} = x^t - \alpha_t(\nabla f(x^t) + e^t),$$

where e^t is the difference between approximate and true gradient.

- The batch size $|\mathcal{B}^t|$ controls size of error e^t.
  - If we sample without replacement from a finite set we get

$$\mathbb{E}[\|e^t\|^2] = \frac{n - |\mathcal{B}^t|}{n} \frac{1}{|\mathcal{B}^t|} \sigma^2,$$

where σ^2 is the variance of the gradient norms.
  - We drive the error to zero as the batch size approaches n.

Approach 1: Batching

- The SG method with a sample $\mathcal{B}^t$ uses iterations

$$x^{t+1} = x^t - \frac{\alpha_t}{|\mathcal{B}^t|} \sum_{i \in \mathcal{B}^t} f_i(x^t).$$

- For a fixed sample size $|\mathcal{B}^t|$, the rate is sublinear.

- But we can grow $|\mathcal{B}^t|$ to achieve a linear rate:
  - Early iterations are cheap like SG iterations.
  - Later iterations can use a Newton-like method.

- Another approach: at some point switch from stochastic to deterministic:
  - Often after a small number of passes.

Stochastic Average Gradient

- Growing $|\mathcal{B}^t|$ eventually requires O(n) iteration cost.
- **Can we have 1 gradient per iteration and only $O(\log(1/\epsilon))$ iterations?**
  - YES! The stochastic average gradient (SAG) algorithm:
    - Randomly select i_t from $\{1, 2, \dots, n\}$ and compute $\nabla f_{i_t}(x^t)$.

$$x^{t+1} = x^t - \frac{\alpha_t}{n} \sum_{i=1}^{n} y_i^t$$

    - **Memory**: $y_i^t = \nabla f_i(x^t)$ from the last t where i was selected.
    - Stochastic variant of earlier increment aggregated gradient (IAG).
  - Key proof idea: $y_i^t \to \nabla f_i(x^*)$ at the same rate that $x^t \to x^*$:
    - So variance of the gradient approximation e^t goes to 0.

Stochastic Average Gradient

- So SAG has a memory

$$\begin{bmatrix} \text{—} & y_1 & \text{—} \\ \text{—} & y_2 & \text{—} \\ & \vdots & \\ \text{—} & y_n & \text{—} \end{bmatrix},$$

  where each y_i keeps track of the last time we randomly picked example i.

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

Convergence Rate of SAG

If each f_i' is $L-$continuous and f is strongly-convex, with $\alpha_t = 1/16L$ SAG has

$$\mathbb{E}[f(x^t) - f(x^*)] \leqslant \left(1 - \min\left\{\frac{\mu}{16L}, \frac{1}{8n}\right\}\right)^t C,$$

where

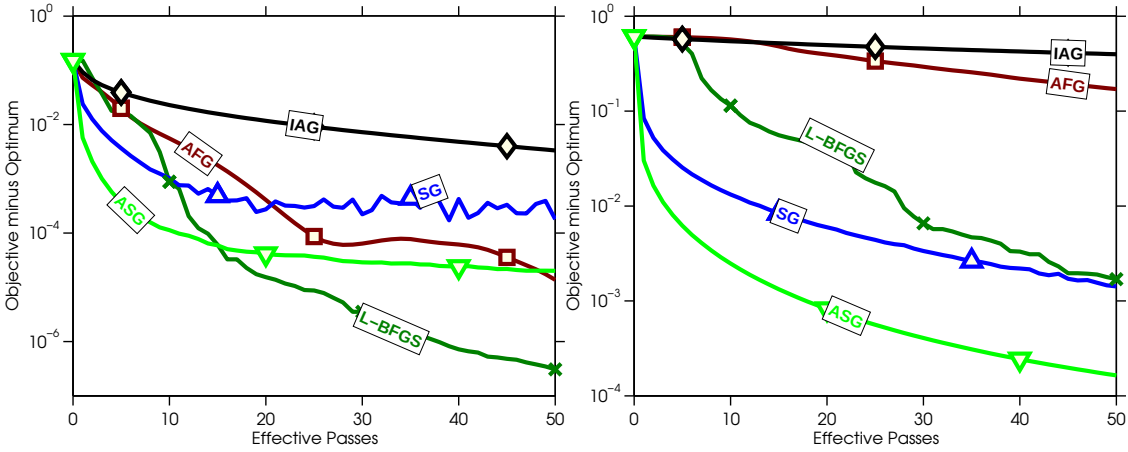
$$C = [f(x^0) - f(x^*)] + \frac{4L}{n}\|x^0 - x^*\|^2 + \frac{\sigma^2}{16L}.$$

- Number of f_i' evaluations to reach ϵ:
  - Stochastic: $O(\frac{L}{\mu}(1/\epsilon))$. (Best when n is enormous)
  - Gradient: $O(n\frac{L}{\mu}\log(1/\epsilon))$.
  - Nesterov: $O(n\sqrt{\frac{L}{\mu}}\log(1/\epsilon))$. (Best when n is small and L/μ is big)
  - SAG: $O(\max\{n, \frac{L}{\mu}\}\log(1/\epsilon))$. (Best when n is big and L/μ is big)

(the L values are again different between algorithms)

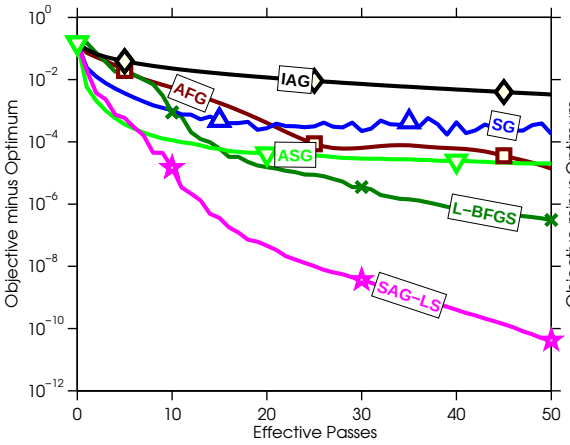
Comparing Deterministic and Stochastic Methods

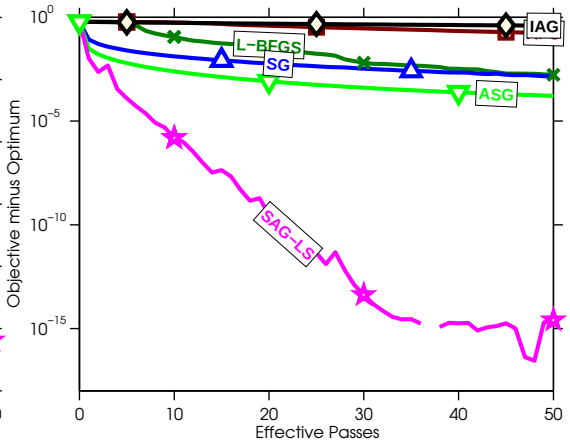
- Two benchmark L2-regularized logistic regression datasets:



SAG Compared to Deterministic/Stochastic Methods

- Two benchmark L2-regularized logistic regression datasets:





SAG Algorithm

- Basic SAG algorithm (maintains $d = \sum_{i=1}^{n} y_i$):
  - Set $d = 0$ and gradient approximation $y_i = 0$ for $i = 1, 2, \ldots, n$.
  - while(1)
    - Sample i from $\{1, 2, \ldots, n\}$.
    - Compute $f_i'(x)$.
    - $d = d - y_i + f_i'(x)$.
    - $y_i = f_i'(x)$.
    - $x = x - \frac{\alpha}{n} d$.

- Iteration cost is $O(d)$, but "lazy updates" allow $O(k)$ with sparse gradients.
- For linear models where $f_i(w) = g(w^T x^i)$, it only require $O(n)$ memory:

$$\nabla f_i(w) = \underbrace{\nabla g(w^T x^i)}_{\text{scalar}} \underbrace{x^i}_{\text{data}}.$$

Discussion of SAG and Beyond

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

$$f_{i_t}\left(x^t - \frac{1}{L}\nabla f_{i_t}(x^t)\right) \leq f_{i_t}(x^t) - \frac{1}{2L}\|\nabla f_{i_t}(x^t)\|^2,$$

    and $\|\nabla f_{i_t}(x^t)\|$ is non-trivial. Costs $O(1)$ for linear models in terms of n and d.
  - Can use $\|x^{t+1} - x^t\|/\alpha = \frac{1}{n}d \approx \|\nabla f(x^t)\|$ 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.

Discussion of SAG and Beyond

- 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.
  - Analyses for infinite data sets.

- Some of the above get rid of the memory...

Stochastic Variance-Reduced Gradient (SVRG)

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

- Start with x_0
- for $s = 0, 1, 2 \ldots$
  - $\nabla f(x_s) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(x_s)$
  - $x^0 = x_s$
  - for $t = 0, 1, 2, \ldots m$
    - Randomly pick $i_t \in \{1, 2, \ldots, n\}$
    - $x^{t+1} = x^t - \alpha_t (\nabla f_{i_t}(x^t) - \underbrace{\nabla f_{i_t}(x_s) + \nabla f(x_s)}_{\text{mean zero}})$.
  - $x_{s+1} = x^t$.

Convergence properties similar to SAG (for m large enough).

(special case of what's known as a "control variate")

$O(d)$ storage at cost of 2 gradients per iteration and n gradients every $O(m)$ iterations.

Outline

Motivation: Multi-Dimensional Polynomial Basis

- Recall using polynomial basis when we only have one features ($x^i \in \mathbb{R}$):

$$\hat{y}^i = w_0 + w_1 x^i + w_2 (x^i)^2.$$

- We can fit these models using a change of basis:

$$\text{If } X = \begin{bmatrix} 0.2 \\ -0.5 \\ 1 \\ 4 \end{bmatrix} \text{ then let } Z = \begin{bmatrix} 1 & 0.2 & (0.2)^2 \\ 1 & -0.5 & (-0.5)^2 \\ 1 & 1 & (1)^2 \\ 1 & 4 & (4^2) \end{bmatrix},$$

and L2-regulairzed least squares solution is

$$w = (Z^T Z + \lambda I)^{-1} Z^T y.$$

- How can we do this when we have a lot of features?

Motivation: Multi-Dimensional Polynomial Basis

- Approach 1: use polynomial basis for each variable:

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

- But this is restrictve:
  - We should allow terms like $x_1^i x_2^i$ that depend on feature interactions.
  - But number of terms in X_{poly} would be huge:
    - Degree-5 polynomial basis has $O(d^5)$ terms:

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

- If n is not too big, we can do this efficiently using the kernel trick.

Equivalent Form of Ridge Regression

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

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

- We showed that the solution is

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

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

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

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

Predictions using Equivalent Form

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

$$\begin{aligned}\hat{y} &= \hat{Z}w \\ &= \hat{Z}Z^T(ZZ^T + \lambda I_n)^{-1}y\end{aligned}$$

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

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

- Key observation behind kernel trick:
  - If we have the K and $\hat{K}$, we don't need the features.

Gram Matrix

- The Gram matrix K is defined by:

$$K = ZZ^T = \begin{bmatrix} — & z_1^T & — \\ — & z_2^T & — \\ & \vdots & \\ — & z_n^T & — \end{bmatrix} \begin{bmatrix} | & | & | \\ z_1 & z_2 & z_3 \\ | & | & | \end{bmatrix}$$

$$= \begin{bmatrix} z_1^T z_1 & z_1^T z_2 & \cdots & z_1^T z_n \\ z_2^T z_1 & z_2^T z_2 & \cdots & z_2^T z_n \\ \vdots & \vdots & \ddots & \vdots \\ z_n^T z_1 & z_n^T z_2 & \cdots & z_n^T z_n \end{bmatrix}$$

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

Polynomial Kernel

- Consider two examples x^i and x^j for a two-dimensional dataset:

$$x^i = (x_1^i, x_2^i), \quad x^j = (x_1^j, x_2^j).$$

- Consider a particular degree-2 basis:

$$z_i = \left((x_1^i)^2, \sqrt{2}x_1^i x_2^i, (x_2^i)^2\right).$$

- We can compute inner product $z_i^T z_j$ without forming z_i and z_j,

$$\begin{aligned} z_i^T z_j &= \begin{bmatrix}(x_1^i)^2 & \sqrt{2}x_1^i x_2^i & (x_2^i)^2\end{bmatrix} z_j \\ &= (x_1^i)^2(x_1^j)^2 + 2x_1^i x_2^i x_1^j x_2^j + (x_2^i)^2(x_2^j)^2 \\ &= (x_1^i x_1^j + x_2^i x_2^j)^2 \qquad \text{(completing the square)} \\ &= ((x^i)^T x^j)^2. \end{aligned}$$

Polynomial Kernel with Higher Degrees

- If we want all degree-4 "monomials", raise it to 4^{th} power:

$$z_i^T z_j = ((x^i)^T x^j)^4,$$

with two variables z_i is weighted version of $(x_1^i)^4, (x_1^i)^3 x_2^i, (x_1^i)^2 (x_2^i)^2, x_1^i (x_2^i)^3, (x_2^i)^4$.

- If you want bias or lower-order terms like x_1^i, add constant inside power:

$$(1 + (x^i)^T x^j)^2 = 1 + 2(x^i)^T x^j + ((x^i)^T x^j)^2$$

$$= \begin{bmatrix} 1 & 2x_1^i & 2x_2^i & (x_1^i)^2 & \sqrt{2}x_1^i x_2^i & (x_2^i)^2 \end{bmatrix} \begin{bmatrix} 1 \\ 2x_1^j \\ 2x_2^j \\ (x_1^j)^2 \\ \sqrt{2}x_1^j x_2^j \\ (x_2^j)^2 \end{bmatrix} = z_i^T z_j,$$

- This pattern still works for any dimension of the x^i.

Kernel Trick

- Using polynomial basis of degree 'p' with the kernel trick:
  - Compute K and $\hat{K}$ which have elements:

$$k(x^i, x^j) = (1 + (x^i)^T x^j)^p, \quad \hat{k}(\hat{x}^i, x^j) = (1 + (\hat{x}^i)^T x^j)^p.$$

  - Make predictions using:

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

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

- Kernel trick:
  - We have kernel function $k(x^i, x^j)$ that gives element (i, j) of K or $\hat{K}$.
  - Skip forming Z and directly form K and $\hat{K}$.
  - 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 $z_i = \exp(-\frac{1}{2}(x^i)^2)v_i$ where $v_i v_j = \exp(x^i x^j)$.
    - 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

$$z_i = \exp\left(-\frac{1}{2}(x^i)^2\right)\begin{bmatrix} 1 & \frac{1}{\sqrt{1!}}x^i & \frac{1}{\sqrt{2!}}(x^i)^2 & \frac{1}{\sqrt{3!}}(x^i)^3 & \cdots \end{bmatrix}.$$

Kernel Trick for Structured Data

- Kernel trick is 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:

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

  - We could convert sentences to features, or define kernel between sentences.
  - For example, "string" kernels:
    - Weighted frequency of common subsequences (dynamic programming).
  - There are also "graph kernels", "image kernels", and so on...

Summary

- Stochastic subgradient methods:
  - $\beta^t v^t$ representation and lazy updates allow sparse datasets.
  - Different step-size strategies and averaging significantly improve performance.
  - Algorithm works with infinite training examples.
- Stochastic average gradient: $O(\log(1/\epsilon))$ iterations with 1 gradient per iteration.
  - SVRG removes the memory requirement.
- Kernel trick: allows working with "similarity" instead of features.
  - Also allows exponential- or infinite-sized feature spaces.

- Next time: when can we use kernel methods, and what are valid kernels?

Bonus Slide: 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

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

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

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

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 $G = X$:

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

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

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