A. Accleration

- inspired by notions of momentum
- 4 papers by Nesterov

A.1. Accelerated proximal gradient method

1. choose initial point $x^{(0)} = x^{(-1)} \in R^n$ (just for notation consistency)
2. repeat

$$v = x^{(k-1)} + \frac{k-2}{k+1}\left(x^{(k-1)} - x^{(k-2)}\right)$$

$$x^{(k)} = prox_{t_k}(v - t_k \nabla g(v))$$

- only some forms of weight satisfy convergence guarantees - this weight is a common example

B. Backtracking line search for acclerated proximal gradient method

B.1. Simple approach

Fix $B < 1$, $t_0 = 1$

At iteration k:

1. start with $t = t_{k-1}$(i.e. where you left off last time, not t_{init})
2. While $g\left(x^+\right) > g(v) + \nabla g(v)^T\left(x^+ - v\right) + \frac{1}{2t}||x^+ - v||_2^2$:

2.1. shrink $t = Bt$

2.2. let $x^+ = prox_t(v - t\nabla g(v))$

3. else keep x^+

Note that this forces you to keep taking decreasing step sizes as k increases.

- this is not the case with usual line search, where we start every iteration at $t = t_{init}$
- (in the limit, the step can

This means that we use less of the gradient descent information in every step

However, note that the weight on the momentum term increases every step as well

So this kind of balances out...?

C. Convergence analysis

For criterion $f(x) = g(x) + h(x)$, such that

1. g is convex, differentiable, dom(G) = R^n, and ∇g is Lipschitz continous with constant L > 0
2. h is conves
3. prox(x) = $argmin_z \left\{\frac{1}{2t}||x - z||_2^2 + h(z)\right\}$ can be evaluated

Theorem:

Accelerated proximal gradient method with fixed step size $t \leqslant 1/L$ satisfies

$$f\left(x^{(k)}\right) - f^* < \frac{2||x^{(0)} - x^*||_2^2}{t(k+1)^2}$$

This is the optimal rate $O\left(1/k^2\right)$ or $O\left(1/\sqrt{\epsilon}\right)$ for first order methods - just using gradient information i.e. no Hessian information, this is the best you can do.

Example: FISTA

To solve lasso, ISTA gave us the update rule

$$B^k = prox_{t_k}\left(B^{(k-1)} - t_k \nabla g\left(B^{(k-1)}\right)\right) = S_{\lambda t}\left(B^{(k-1)} + t_k X^T(y - XB)\right)$$

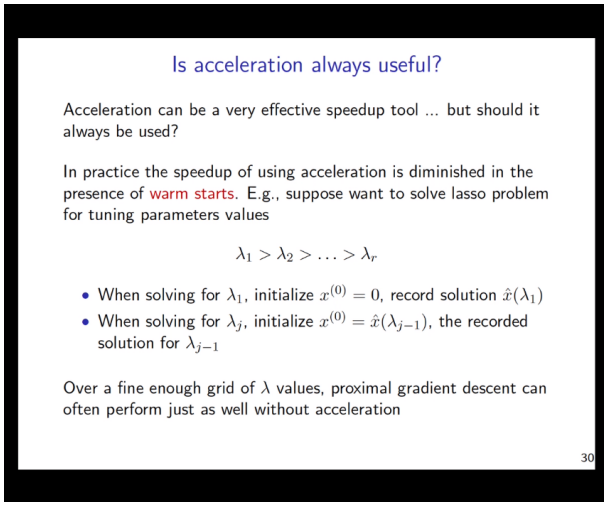
Applying acceleration gives Fast ISTA = FISTA

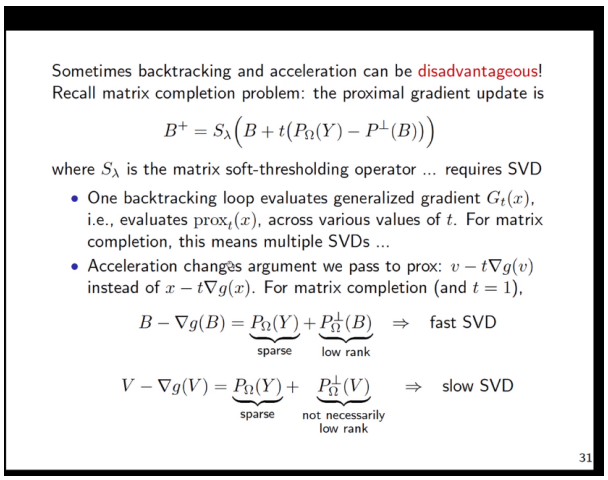
where

$$v = B^{(k-1)} + \frac{k-2}{k+1}\left(B^{(k-1)} - B^{(k-2)}\right)$$

$$B^k = S_{\lambda t}\left(v^{(k-1)} + t_k X^T(y - Xv)\right)$$

Comment: in general momentum is not as well understood as it should be. We don't really understand why this works beyond the intuition that if you believe that the update you made was in the right direction, you should continue in the same dir, and that this is helpful when you get closer to the optimm because tehen the gradient will be small





- $P(V)$is not low rank because rank is not a convex function!
- You can add two low rank matrices and get a matrix of higher rank!
- this is why you try and convexify it in the first place

This makes each iteration (k) more expensive. Maybe the number of iterations to convergence is lower, but each individual iteration may be so much more expensive that it is no longer worth it

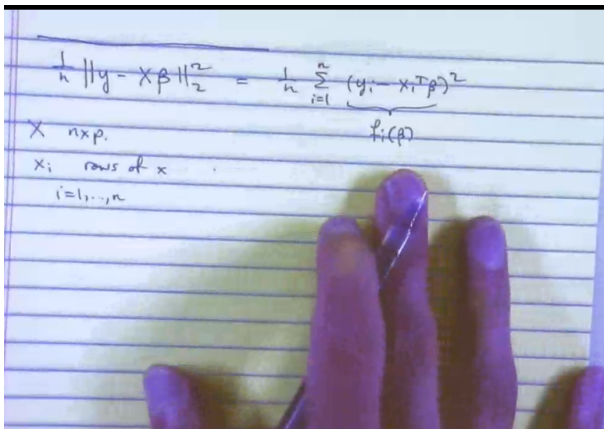
---

A. Stochastic Gradient Descent

Consider minimizing an average of functions

$$min_x \frac{1}{m}\sum_{i=1}^{m} f_i(x)$$

An example is mean squared error, which can be rewritten as the mean of the following function, applied per data point



Then gradient descent would repeat

$$x^{(k)} = x^{(k-1)} - t_k \cdot \frac{1}{m} \sum_{i=1}^{m} \nabla f_i\left(x^{(k-1)}\right)$$

In comparison, SGD, or incremental gradient descent, repeats

$$x^{(k)} = x^{(k-1)} - t_k \cdot \nabla f_{i_k}\left(x^{(k-1)}\right)$$

where $i_k \in \{1, \ldots, m\}$ is some chosen index at iteration k

There are two rules for choosing index i_k.

1. Randomized: choose it uniformly at random
2. Cyclic: go through the list in sequence and wrap around

In general, ranfomized rules are easier to analyze because you can reason about its expected value

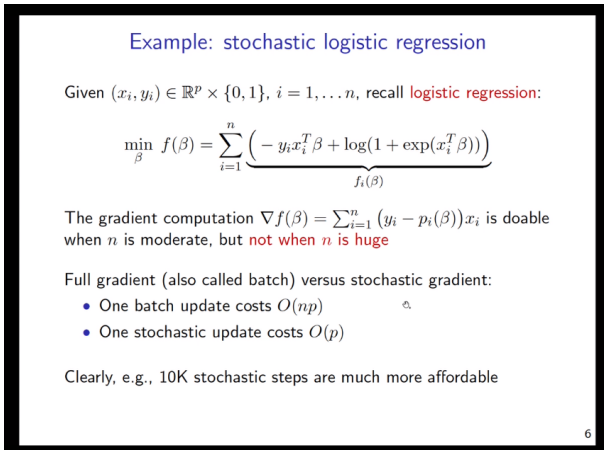
Randomized rule is more common in practice. Note that for hte randomized rule

$$E[\nabla f_{i_k}(x)] = \frac{1}{m} \sum_{i=1}^{m} \nabla f_i(x)$$

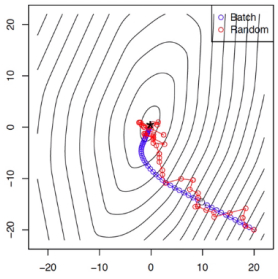
so SGD is an unbiased estimate of the gradient at each step.

The appeal of SGD:

1. iteration cost is independent of m (number of functions)
    - the cost of computing only depends on p
2. big memory useage savings
    - dont need to hold all the data in memory - just one data point at a time



Small example with $n = 10$, $p = 2$ to show the "classic picture" for batch versus stochastic methods:



Blue: batch steps, $O(np)$
Red: stochastic steps, $O(p)$

Rule of thumb for stochastic methods:

- generally thrive far from optimum
- generally struggle close to optimum

7

Note that this **is not strictly a gradient descent method!!**

- convention has come to be this way though

A.1. Step sizes

In the optimization community, the standard in SGD is to use diminishing step sizes, e.g. $t_k = 1/k$

- this may not be true for the ML community!

Why not fixed step sizes?

Step sizes

Standard in SGD is to use diminishing step sizes, e.g., $t_k = 1/k$, for $k = 1, 2, 3, \ldots$

Why not fixed step sizes? Here's some intuition. Suppose we take cyclic rule for simplicity. Set $t_k = t$ for m updates in a row, we get:

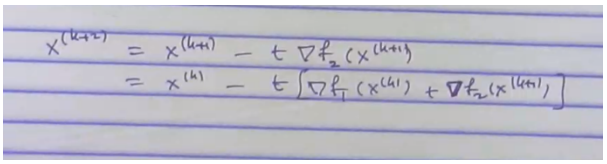
$$x^{(k+m)} = x^{(k)} - t \sum_{i=1}^{m} \nabla f_i(x^{(k+i-1)})$$

Meanwhile, full gradient with step size t would give:

$$x^{(k+1)} = x^{(k)} - t \sum_{i=1}^{m} \nabla f_i(x^{(k)})$$

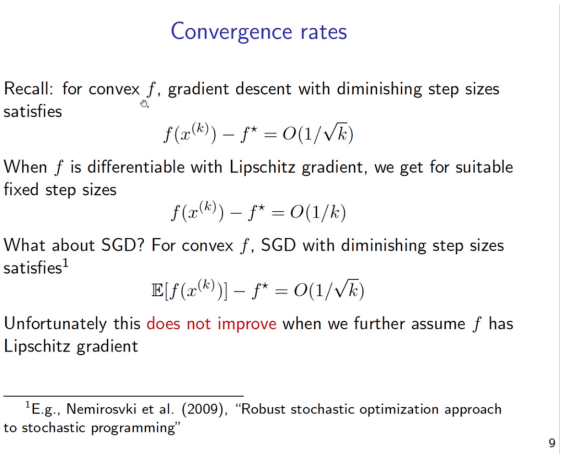
The difference here: $t \sum_{i=1}^{m} [\nabla f_i(x^{(k+i-1)}) - \nabla f_i(x^{(k)})]$, and if we hold t constant, this difference will not generally be going to zero

8



- SGD does not get arbitrarily close to gradient descent if we keep t constant
  - by diminishing the step size, we make this difference smaller

A.2. Convergence



Convergence rates

Recall: for convex f, gradient descent with diminishing step sizes satisfies

$$f(x^{(k)}) - f^\star = O(1/\sqrt{k})$$

When f is differentiable with Lipschitz gradient, we get for suitable fixed step sizes

$$f(x^{(k)}) - f^\star = O(1/k)$$

What about SGD? For convex f, SGD with diminishing step sizes satisfies[1]

$$\mathbb{E}[f(x^{(k)})] - f^\star = O(1/\sqrt{k})$$

Unfortunately this does not improve when we further assume f has Lipschitz gradient

[1] E.g., Nemirosvki et al. (2009), "Robust stochastic optimization approach to stochastic programming"

9

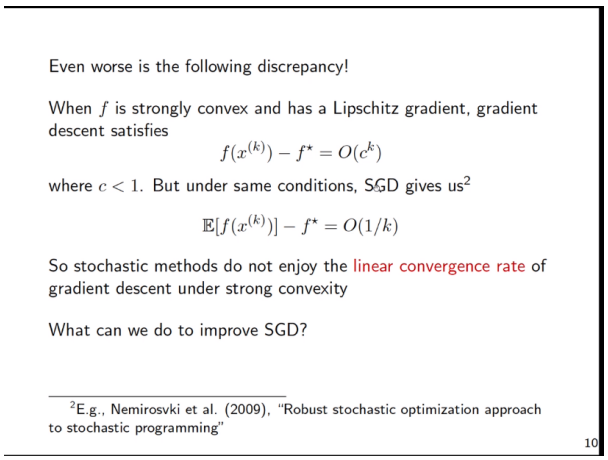
With gradient descent:

1. if we dont know that the gradient is **Lipschitz**, $f\left(x^{(k)}\right) - f^* = O\left(1/\sqrt{k}\right)$

2. if we do that the gradient is Lipschitz, $f\left(x^{(k)}\right) - f^* = O(1/k)$

With SGD, it only satisifes $E\left[f\left(x^{(k)}\right)\right] - f^* = O\left(1/\sqrt{k}\right)$ even if the gradient is Lipschitz - does not improve unlike gradient descent!

- it is stuck in this slow convergence regime



Under **strong convexity**, GD gives you 'linear convergence' (i.e. exponential convergence) But SGD only gives you O(1/k).

It only matches GD in the most basic case, when you know nothing about the function.

Questions
1. can we improve SGD?
2. who cares?

A.3. Improving SGD with Mini-batches

Pick a random subset $I_k \subset \{1, ..., m\}$ of size $|I_k| = b \ll m$
Then repeat

$$x^{(k)} = x^{(k-1)} - t_k \cdot \frac{1}{b} \sum_{i \in I_k} \nabla f_i\left(x^{(k-1)}\right)$$

This approximates the full gradient by an unbiased estimate

$$E\left[\frac{1}{b} \sum_{i \in I_k} \nabla f_i(x)\right] = \nabla f(x)$$

This reduces the **variance** of our gradient estimate by 1/b but each calcluation is also b times more expensive.

A.3.1. Q: Is this tradeoff worth it?

Back to logistic regression, let's now consider a regularized version:

$$\min_{\beta \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^{n} \Big(- y_i x_i^T \beta + \log(1 + e^{x_i^T \beta}) \Big) + \frac{\lambda}{2} \|\beta\|_2^2$$

Write the criterion as

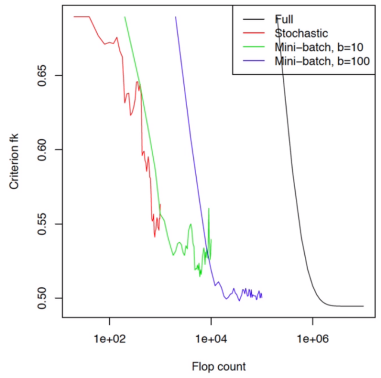
$$f(\beta) = \frac{1}{n} \sum_{i=1}^{n} f_i(\beta), \quad f_i(\beta) = -y_i x_i^T \beta + \log(1 + e^{x_i^T \beta}) + \frac{\lambda}{2} \|\beta\|_2^2$$

The gradient computation is $\nabla f(\beta) = \sum_{i=1}^{n} \big(y_i - p_i(\beta)\big) x_i + \lambda \beta$.
Comparison between methods:

- One batch update costs $O(np)$
- One mini-batch update costs $O(bp)$
- One stochastic update costs $O(p)$

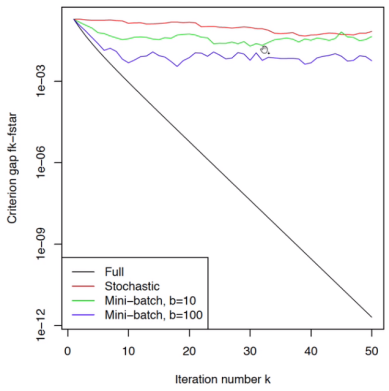
12

What's happening? Now let's parametrize by flops:



If you reall care about progress per unit computation, it may be the case that SGD is the best - though it may well bounce around

Finally, looking at suboptimality gap (on log scale):



With regularization, the criterion is strongly convex, so we get linear convergence, but the others are a bit pathetic...

End of the story?

Short story:

- SGD can be super effective in terms of iteration cost, memory
- But SGD is slow to converge, can't adapt to strong convexity
- And mini-batches seem to be a wash in terms of flops (though they can still be useful in practice)

Is this the end of the story for SGD?

For a while, the answer was believed to be yes. Slow convergence for strongly convex functions was believed inevitable, as Nemirovski and others established matching lower bounds ... but this was for a more general stochastic problem, where $f(x) = \int F(x, \xi)\, dP(\xi)$

New wave of "variance reduction" work shows we can modify SGD to converge much faster for finite sums (more later?)

16

In practice, mini batches may be useful because of architectural constraints - there may be a communication cost overhead from reading data points one at a time

- implementation problems, not statistical

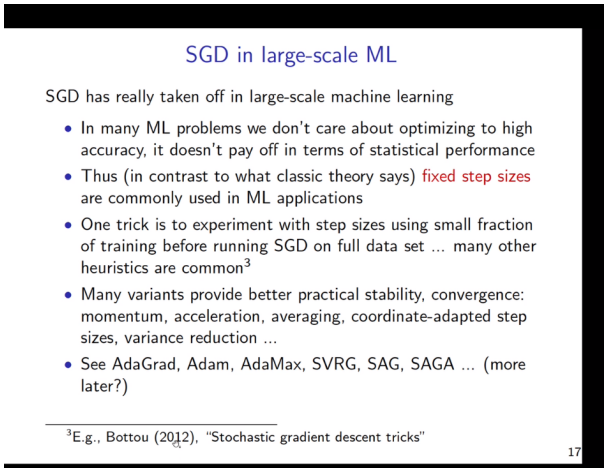
For a while, this was believed to be the end fo the story - but it was shown that these boudns were true for a general class of stochastic problem
In many ML instances, we are dealing typically with means or discrete sums, for which there are better bounds! Modifications to SGD help here - you can actually achieve strong convexity for strongly convex criterion functions

- SVRG, SAG, SAGA - all variance reduction techniques
  - use some info from earlier gradients

A.3.2. who cares?

In ML people often don't care for the optimum - it is not known if the optimum parameter set is significantly better at a prediction task than something just close/near it

- so people weigh up the per-compute cost for large scale problems and use SGD
- they also use fixed step sizes typically for this reason

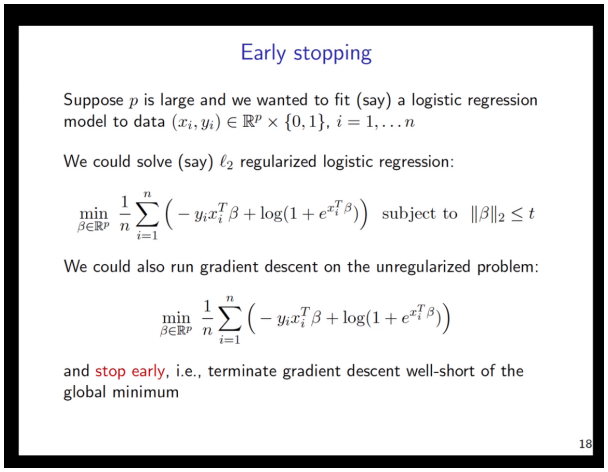


Adam, AdaGrad, AdaMax

- provide faster convergence for SGD by using
  - momentum
  - coordinate adaptive step sizes

B. Early stopping

Strange to optimization but used in machine learning a lot



- Stop very early - far away from the minimum
- This is probably when you think the actual optimum is not very helpful/useful to you

Consider the following, for a very small constant step size ϵ:

- Start at $\beta^{(0)} = 0$, solution to regularized problem at $t = 0$
- Perform gradient descent on unregularized criterion

$$\beta^{(k)} = \beta^{(k-1)} - \epsilon \cdot \frac{1}{n} \sum_{i=1}^{n} (y_i - p_i(\beta^{(k-1)}))x_i, \quad k = 1, 2, 3, \ldots$$

  (we could equally well consider SGD)
- Treat $\beta^{(k)}$ as an approximate solution to regularized problem with $t = \|\beta^{(k)}\|_2$

This is called early stopping for gradient descent. Why would we ever do this? It's both more convenient and potentially much more efficient than using explicit regularization

19

- Think of the intermediate paths as being estimates of the regularization parameter
- Useful if for e.g. you dont even know what param to use

Lots left to explore

- Connection holds beyond logistic regression, for arbitrary loss
- In general, the grad descent path will not coincide with the ℓ_2 regularized path (as $\epsilon \to 0$). Though in practice, it seems to give competitive statistical performance
- Can extend early stopping idea to mimick a generic regularizer (beyond ℓ_2)[4]
- There is a lot of literature on early stopping, but it's still not as well-understood as it should be
- Early stopping is just one instance of implicit or algorithmic regularization ... many others are effective in large-scale ML, they all should be better understood

[4] Tibshirani (2015), "A general framework for fast stagewise algorithms"

21

- **Dropout** also implictly performs regularization