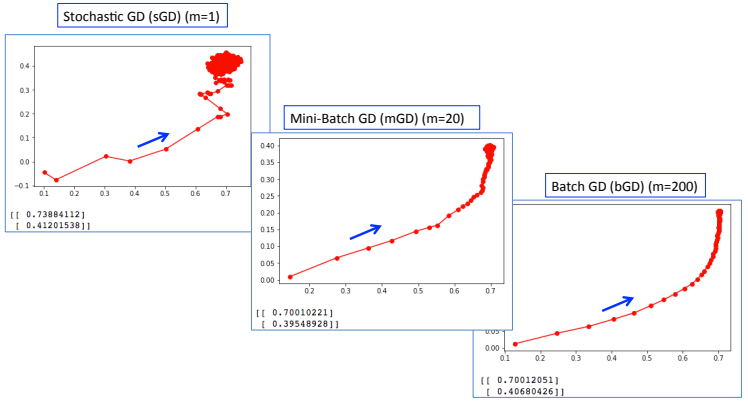
Optimization
(which optimizer?)

Optimization
(which subsets?)



Many beautiful slides (marked as Ho) were taken from this webstie.

Joyce Ho — Research | Publications | Teaching

CS 584: Big Data Analytics

Course Description

The course covers scalable machine learning and data mining algorithms for large/complex data. Topics include large-scale optimization techniques, hashing, recommendation systems, and tensor factorization. This will be structured as a seminar course with emphasis on public data sets such as Kaggle competitions, MovieLens, and various healthcare datasets. There will be introductory lectures that set the context and provide reviews of relevant material.
Instructor: Joyce Ho
Office Hours: Tu/Th 1-4 pm at MSC W414

Course Prerequisites

Graduate Data Mining (CS 570) and familiarity with Python, Matlab, or R.

2.3 The Approximation–Estimation–Optimization Tradeoff

This decomposition leads to a more complicated compromise. It involves three variables and two constraints. The constraints are the maximal number of available training example and the maximal computation time. The variables are the size of the family of functions $\mathcal{F}$, the optimization accuracy ρ, and the number of examples n. This is formalized by the following optimization problem.

$$\min_{\mathcal{F},\rho,n} \quad \mathcal{E} = \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{opt}} \quad \text{subject to} \begin{cases} n & \leq & n_{\max} \\ T(\mathcal{F},\rho,n) & \leq & T_{\max} \end{cases} \tag{1.3}$$

Table 1: Typical variations when $\mathcal{F}$, n, and ρ increase.

		$\mathcal{F}$	n	ρ
$\mathcal{E}_{\text{app}}$	(approximation error)	↘		
$\mathcal{E}_{\text{est}}$	(estimation error)	↗	↘	
$\mathcal{E}_{\text{opt}}$	(optimization error)	⋯	⋯	↗
T	(computation time)	↗	↗	↘

Bottom and Bousquet (2011). The Tradeoffs of Large-Scale Learning. In Optimization for Machine Learning (pp. 351–368).

Gradient Descent Reformulated

$$w^+ = w - \gamma \frac{1}{n} \sum_i \nabla_w L(f_w(x_i), y_i) \quad \nabla E_n(f_w)$$

learning rate or gain

- True gradient descent is a batch algorithm, slow but sure
- Under sufficient regularity assumptions, initial estimate is close to the optimal and gain is sufficiently small, there is linear convergence

Stochastic Optimization Motivation

- Information is redundant amongst samples
- Sufficient samples means we can afford more frequent, noisy updates
- Never-ending stream means we should not wait for all data
- Tracking non-stationary data means that the target is moving

Stochastic Optimization

- Idea: Estimate function and gradient from a small, current subsample of your data and with enough iterations and data, you will converge to the true minimum
- Pro: Better for large datasets and often faster convergence
- Con: Hard to reach high accuracy
- Con: Best classical methods can't handle stochastic approximation
- Con: Theoretical definitions for convergence not as well-defined

Stochastic Gradient Descent (SGD)

- Randomized gradient estimate to minimize the function using a single randomly picked example

  Instead of ∇f, use $\tilde{\nabla} f$, where $E[\tilde{\nabla} f] = \nabla f$

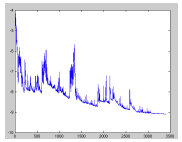
- The resulting update is of the form:

  $$w^+ = w - \gamma \nabla_w L(f_w(x_i, y_i))$$

- Although random noise is introduced, it behaves like gradient descent in its expectation

SGD Algorithm

Randomly initialize parameter w and learning rate γ
while *Not Converged* **do**
  Randomly shuffle examples in training set
  for $i = 1, \cdots, N$ **do**
    $w^+ = w - \gamma \nabla_w L(f_w(x_i, y_i))$
  end
end



The Benefits of SGD

- Gradient is easy to calculate ("instantaneous")
- Less prone to local minima
- Small memory footprint
- Get to a reasonable solution quickly
- Works for non-stationary environments as well as online settings
- Can be used for more complex models and error surfaces

Mini-batch Stochastic Gradient Descent

- Rather than using a single point, use a random subset where the size is less than the original data size

  $$w^+ = w - \gamma \frac{1}{|S_k|} \sum_{i \in S_k} \nabla_w L(f_w(x_i, y_i)), \text{where } S_k \subseteq [n]$$

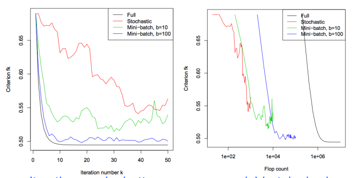
- Like the single random sample, the full gradient is approximated via an unbiased noisy estimate
- Random subset reduces the variance by a factor of $1/|S_k|$, but is also $|S_k|$ times more expensive

Update costs:

- Batch: $O(nd)$
- Stochastic: $O(d)$
- Mini-batch: $O(|S_k|d)$

Batch is doable if n is moderate but not when n is huge

Example: n=10,000, d=20



Iterations make better progress as mini-batch size is larger but also takes more computation time

http://stat.cmu.edu/~ryantibs/convexopt/lectures/25-fast-stochastic.pdf

CS 584 [Spring 2016] - Ho

SGD Recommendations

- Randomly shuffle training examples
  - Although theory says you should randomly pick examples, it is easier to make a pass through your training set sequentially
  - Shuffling before each iteration eliminates the effect of order
- Monitor both training cost and validation error
  - Set aside samples for a decent validation set
  - Compute the objective on the training set and validation set (expensive but better than overfitting or wasting computation)

CS 584 [Spring 2016] - Ho

Bottou, L. (2012). Stochastic Gradient Descent Tricks. Neural Networks Tricks of the Trade.

SGD Recommendations (2)

- Check gradient using finite differences
  - If computation is slightly incorrect can yield erratic and slow algorithm
  - Verify your code by slightly perturbing the parameter and inspecting differences between the two gradients
- Experiment with the learning rates using small sample of training set
  - SGD convergence rates are independent from sample size
  - Use traditional optimization algorithms as a reference point

CS 584 [Spring 2016] - Ho

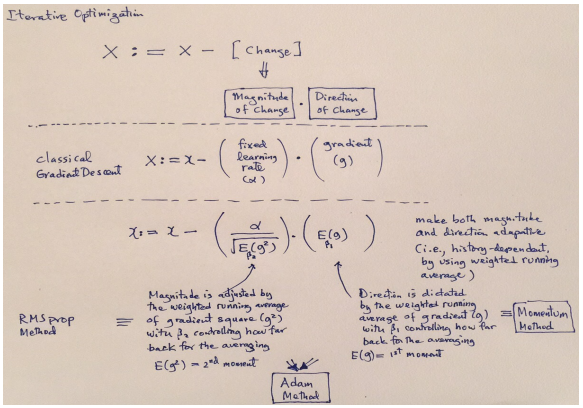
SGD Recommendations (3)

- Leverage sparsity of the training examples
  - For very high-dimensional vectors with few non zero coefficients, you only need to update the weight coefficients corresponding to nonzero pattern in x
- Use learning rates of the form $\gamma_t = \gamma_0(1 + \gamma_0 \lambda t)^{-1}$
  - Allows you to start from reasonable learning rates determined by testing on a small sample
  - Works well in most situations if the initial point is slightly smaller than best value observed in training sample

CS 584 [Spring 2016] - Ho

Optimization

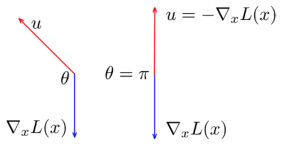
(which direction?)



Method of Steepest Descent

$$\lim_{\alpha \to 0} L(\mathrm{x}+\alpha \mathrm{u}) = u^T \nabla_{\mathrm{x}} L(x).$$

$$u^T \nabla_{\mathrm{x}} L(x) = |u| \cdot |\nabla_{\mathrm{x}} L(x)| \cos\theta,$$



(s/m/b) GD

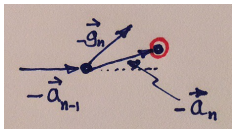
Tf.train.GradientDescentOptimizer

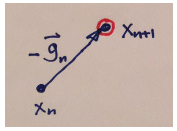
Momentum

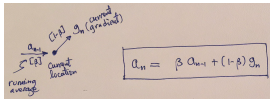
$$x = \mathrm{x} - \alpha \nabla_{\mathrm{x}} L(x).$$

$$\mathrm{x} = \mathrm{x} - \mathrm{u}_s$$

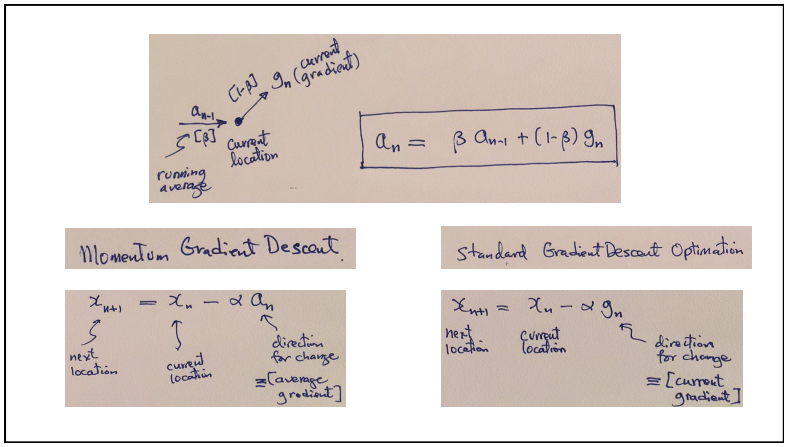
$$\mathrm{u}_s = \gamma u_{s-1} + \alpha \nabla_{\mathrm{x}} L(x).$$

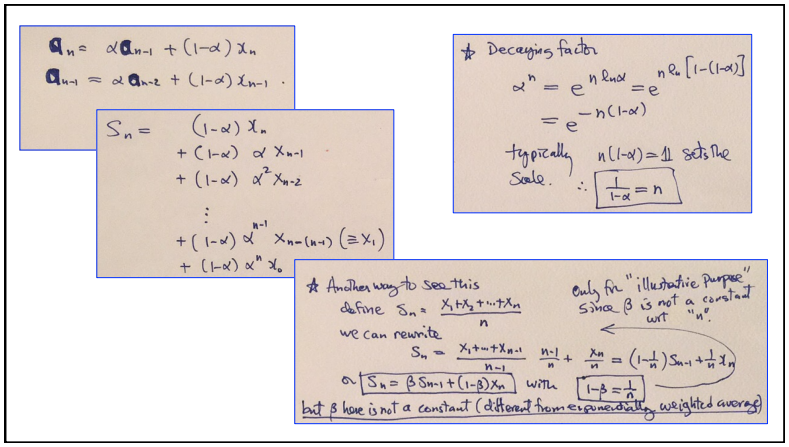






Tf.train.MomentumOptimizer





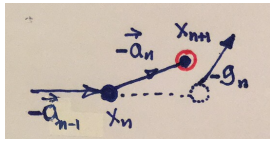
Nesterov Accelerated Gradient (NAS)

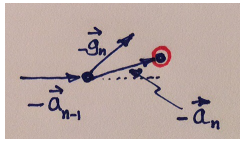
$x = x - u_s$

$u_s = \gamma u_{s-1} + \alpha \nabla_x L(x - \gamma u_{s-1}).$

Compared with Momentum

$u_s = \gamma u_{s-1} + \alpha \nabla_x L(x).$





$\vec{a}_n = \beta\, \vec{a}_{n-1} + (1-\beta)\, \vec{g}_n$

but $\vec{g}_n$ is not evaluated @ x_n, but @ $(x_n - \beta \vec{a}_{n-1})$

Tf.train.NAGOptimizer

Annealing and Learning Rate Schedules

Adagrad

The Adagrad algorithm adjusts the learning rate for each parameter. So far, we have been denoting the parameters of the loss function as x. Note that x is actually a large number of parameters, each of which is being updated with the same learning rate. Let x_i denote one of the parameters and let g_i^s denote the gradient for x_i at step s. For steps $0,1,\ldots,S-1$ we have a corresponding series of gradients $g_i^0, g_i^1, \ldots, g_i^s$.

$$G = \left(g_i^0\right)^2 + \left(g_i^1\right)^2 + \ldots + \left(g_i^{S-1}\right)^2$$

$$x_i = x_i - \frac{\alpha}{\frac{1}{G^2}}\, g_i^s.$$

The α term is the global learning rate, which gets adapted for each parameter based on the previous gradients. It must also be noted that as G accumulates, the learning rate slows down for each parameter and eventually no progress can be made, which is a weakness of Adagrad.

Tf.train.GradientAdagradptimizer

RMSProp

The RMSProp algorithm improves on the Adagarad algorithm's weakness of completely halted progress beyond a certain number of iterations. The intuition here is to use a window of fixed size over the gradients computed at each step rather than use the full set of gradients. That is, compute G over only the past W steps. Now, it's conceptually equivalent but computationally cheaper to treat the computation of

$$G=\frac{\left(g_i^{s-W}\right)^2+\left(g_i^{s-W+1}\right)^2+\ldots+\left(g_i^{s-1}\right)^2}{W}$$

as the accumulation of exponentially decaying average of square of gradients rather than store all values of

$$g_i^{-W}, g_i^{-W+1}, \ldots, g_i^{-2} g_i^{-1}$$

and compute G at each step. That is, we compute

$$E\left[\left(g_i\right)^2\right]^s=\rho E\left[\left(g_i\right)^2\right]^{s-1}+(1-\rho)\left(g_i^s\right)^2$$

(Rho controls W,
Rho=0.9 W=10
Rho=0.99 W=100
…)

where ρ is the decay.

Now, note that in Adagrad we were computing the update as

$x_i=x_i-\frac{\alpha}{G^{\frac{1}{2}}}g_i^s$. Consider the value of $G^{\frac{1}{2}}$. We can see that this is simply the root-mean-square of g_i, that is

RMSProp

The RMSProp algorithm improves on the Adagarad algorithm's weakness of completely halted progress beyond a certain number of iterations. The intuition here is to use a window of fixed size over the gradients computed at each step rather than use the full set of gradients. That is, compute G over only the past W steps. Now, it's conceptually equivalent but computationally cheaper to treat the computation of

$$G=\frac{\left(g_i^{s-W}\right)^2+\left(g_i^{s-W+1}\right)^2+\ldots+\left(g_i^{s-1}\right)^2}{W}$$

$$g_i^{-W}, g_i^{-W+1}, \ldots, g_i^{-2} g_i^{-1}$$

$$E\left[\left(g_i\right)^2\right]^s=\rho E\left[\left(g_i\right)^2\right]^{s-1}+(1-\rho)\left(g_i^s\right)^2$$

(Don't store, but running average)

(Rho controls W,
Rho=0.9 W=10
Rho=0.99 W=100
…)

Now, note that in Adagrad we were computing the update as

$x_i=x_i-\frac{\alpha}{G^{\frac{1}{2}}}g_i^s$. Consider the value of $G^{\frac{1}{2}}$. We can see that this is simply the root-mean-square of g_i, that is

$$RMS\left[g_i\right]=G^{\frac{1}{2}}=\sqrt{E\left[\left(g_i\right)^2\right]^s}.$$

$$x_i=x_i-\frac{\alpha}{RMS\left[g_i\right]}g_i^s.$$

Adadelta

The intuition behind Adadelta is to consider whether the unit of the parameter and the update to the parameter is the same. The author of the Adadelta argues that this is not the same in the case of any first order methods like steepest descent (but is the same in the case of second order methods like Newton's method). In order to fix this issue, the proposed update rule of Adadelta is

$$x_i=x_i-\frac{RMS\left[\Delta x_i\right]^{s-1}}{RMS\left[g_i\right]^s}g_i^s$$

where $RMS\left[\Delta x\right]^{s-1}$ is the root-mean-square of the actual updates to x. Note that $RMS\left[\Delta x_i\right]^{s-1}$ lags behind $RMS[g_i]^s$ by one step.

Adam

Adam computes the updates by maintaining the exponentially weighted averages of both g_i and $(g_i)^2$ for each parameter (denoted by the subscript i). The update rule for Adam is

$$x_i=x_i-\frac{\alpha}{E\left[\left(g_i^{s-1}\right)\right]^2}E\left[g_i^{s-1}\right].$$

It is important to note that $E\left[g_i^{s-1}\right]$ and $E\left[\left(g_i^{s-1}\right)^2\right]$ are biased towards zero in the initial steps for small decay rates (there are two decay rates here—one for $E\left[g_i^{s-1}\right]$ and one for $E\left[\left(g_i^{s-1}\right)^2\right]$—which we denote by ρ_1 and ρ_2 respectively). This bias can be corrected by computing

$$E\left[g_i^{s-1}\right]=\frac{E\left[g_i^{s-1}\right]}{1-\rho_1}$$

and

$$E\left[\left(g_i^{s-1}\right)^2\right]=\frac{E\left[\left(g_i^{s-1}\right)^2\right]}{1-\rho_2}$$

How would these algorithms behave on complicated loss function landscape?