Optimization for Machine Learning

James Martens (DeepMind)

Overview

Topics:

- Gradient descent
- Momentum methods
- 2nd-order methods
- Stochastic optimization

Motivation

- Numerical optimization methods enable models to learn from data by adapting their parameters
  - They are the basic engine behind most modern machine learning techniques

- They solve the problem of minimizing some (given) objective function that quantifies the performance of the model
  - E.g. prediction error, mistakes on some task, etc

- Usually work by making small incremental changes to parameters that slowly decrease objective towards (local) minimum
  - This strategy works only if the objective functions are nicely behaved (smooth, etc)

Notation

- Parameters: θ
- Objective function : $h(\theta)$
- Goal: $\theta^* = \arg\min_{\theta} h(\theta)$

Gradient descent

Definition

- Basic gradient descent iteration:

$$\theta_{k+1} = \theta_k - \alpha_k \nabla h(\theta_k)$$

Step size: α_k

(aka “learning rate”)

Gradient: $\nabla h(\theta) = \begin{bmatrix} \frac{\partial h(\theta)}{\partial [\theta]_1} \\ \frac{\partial h(\theta)}{\partial [\theta]_2} \\ \vdots \\ \frac{\partial h(\theta)}{\partial [\theta]_n} \end{bmatrix}$

Gradient Descent

Intuition / motivation

$$\theta_{k+1} = \theta_k - \alpha_k \nabla h(\theta_k)$$

Why should this work?

- Gradient direction gives greatest reduction in $h(\theta)$ per unit of change* in θ
- Formally: $\frac{-\nabla h}{\|\nabla h\|} = \lim_{\epsilon \to 0} \frac{1}{\epsilon} \arg\min_{d:\|d\| \leq \epsilon} h(\theta + d)$
- If $h(\theta)$ is relatively "smooth", $\nabla h(\theta)$ will keep pointing down-hill as long as we don't go too far from the current θ

Gradient Descent

Intuition / motivation

Motivation from local approximations:

- 1st-order Taylor series for $h(\theta)$ around current θ is:

$$h(\theta + d) \approx h(\theta) + \nabla h(\theta)^\top d$$

- For small enough d this will be a reasonable approximation
- Gradient update computed by minimizing this within a sphere of radius r:

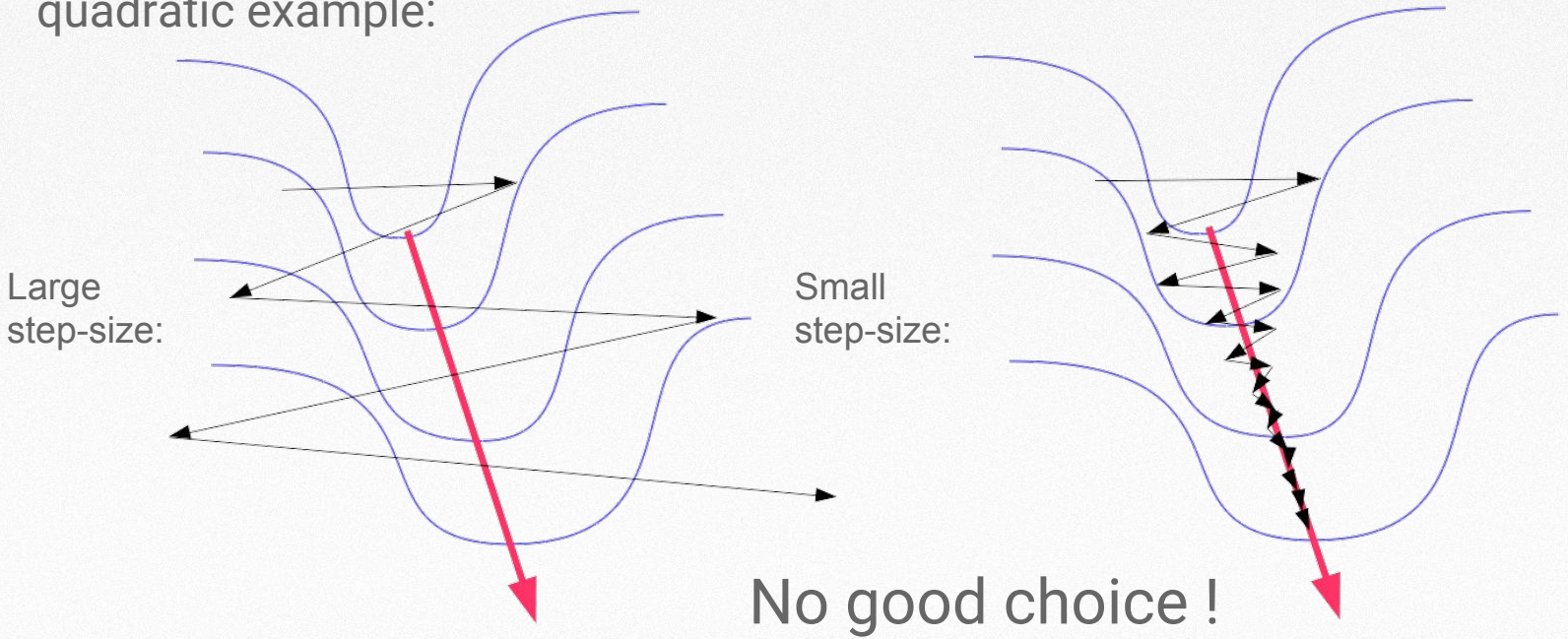
$$-\alpha \nabla h(\theta) = \arg\min_{d : \|d\| \leq r} \left(h(\theta) + \nabla h(\theta)^\top d \right)$$

where $r = \alpha \|\nabla h(\theta)\|$

Problems with gradient descent

Failure case

- Standard failure case for gradient descent is a simple two-dimensional quadratic example:



No good choice !

Problems with gradient descent

Failure case

- Convergence can be slow for functions whose curvature varies wildly depending on which direction you point

- There is no “sweet-spot” step-size to use. You either have:
  - Large oscillations along directions of high curvature causing divergence
  - Very slow progress along directions of small curvature

Problems with gradient descent

Technical explanation of failure

- Gradient descent minimizes the following primitive local ***2nd-order*** approximation to $h(\theta)$:

$$h(\theta + d) \approx h(\theta) + \nabla h(\theta)^\top d + \frac{1}{2} d^\top H(\theta) d$$

$$\approx h(\theta) + \nabla h(\theta)^\top d + \frac{1}{2} d^\top (LI) d = h(\theta) + \nabla h(\theta)^\top d + \frac{L}{2} \|d\|^2$$

Whose solution is: $-\frac{1}{L} \nabla h(\theta) = \arg\min_d \left(h(\theta) + \nabla h(\theta)^\top d + \frac{L}{2} \|d\|^2 \right)$

- LI is a very conservative / pessimistic approximation to $H(\theta)$ that treats all directions as having the same (very high) curvature.
  - This helps explain why it struggles on problems where curvature varies a lot

Some standard technical assumptions

- $h(\theta)$ has Lipschitz continuous derivatives (i.e. is "Lipschitz smooth"):

$$\|\nabla h(\theta) - \nabla h(\theta')\| \leq L\|\theta - \theta'\|$$ (an **upper bound** on the curvature)

Intuitively: the gradient doesn't change too fast, implying that the gradient will remain a descent direction in a small local neighborhood around the current θ

- $h(\theta)$ is strongly convex (perhaps only near some local min):

$$h(\theta + d) \geq h(\theta) + \nabla h(\theta)^\top d + \frac{\mu}{2}\|d\|^2$$ (a **lower bound** on the curvature)

- And *for now:* Gradients and other quantities are computed exactly (i.e. **not** stochastic)

Convergence theory

Upper bounds for gradient descent

If $h(\theta)$ is Lipschitz smooth and (locally) strongly convex, and θ^* is the (local) minimizer, gradient descent satisfies the upper bound:

$$h(\theta_k) - h(\theta^*) \leq \frac{L}{2}\left(\frac{\kappa - 1}{\kappa + 1}\right)^{2k} \|\theta_0 - \theta^*\|^2 \quad \text{for} \quad \alpha_k = \frac{2}{L + \mu}$$

where, $\kappa = \dfrac{L}{\mu}$ is a "condition number" = ratio of highest curvature to lowest curvature.

Number of iterations to achieve $h(\theta_k) - h(\theta^*) \leq \epsilon$: $\quad k \in \mathcal{O}\left(\kappa \log \frac{1}{\epsilon}\right)$

Convergence theory

Warnings, caveats, etc.

- These bounds must work for *all* objective functions in the given class
  - this includes *worst-case* examples
  - real problems are seldom worst-case
  - thus bounds are often pessimistic / unrealistic
- They often do not take into account all the useful structure in the real objective.
- For example, the condition number ignores:
  - clustered eigenvalues in Hessian
  - low-curvature directions that are completely flat (i.e. not important to optimize)

Convergence theory

Warnings, caveats, etc.

- Bounds only accurately describe asymptotic performance
  - And often we stop before asymptotics “kick-in”. Either to prevent overfitting, or because we have a fixed computational budget
  - Early-stage optimization can behave much different than late-stage (travelling in a roughly consistent direction vs bouncing around local min)

- Provide no global guarantees for non-convex objectives

- The design/choice of an optimizer should always be informed by practice more than anything else. *But*, good theory can help guide the way and build intuitions.

Momentum

Motivation and intuition

- A very simple way to “accelerate” gradient descent (and other optimizers)
- Motivation:
  - the direction of descent (gradient) can vary with each iteration
  - some directions may flip back and forth between pointing uphill and downhill
  - we saw this behavior for gradient descent applied to “failure case” example
- Solution:
  - accelerate along directions that point down-hill consistently
- How?
  - treat optimizer like as a “ball” rolling around the “surface” defined by the objective function - i.e. let it accumulate velocity like physical objects do

Gradient descent with momentum

Defining equations

- Classical Momentum:

$$v_{k+1} = \eta_k v_k - \alpha_k \nabla h(\theta_k)$$
$$\theta_{k+1} = \theta_k + v_{k+1}$$

Learning rate: α_k

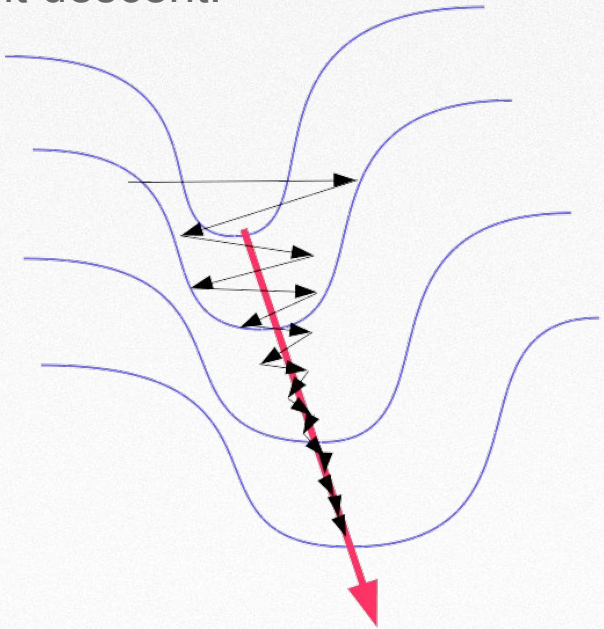
Friction constant: η_k

- Nesterov's version (aka Nesterov's accelerated gradient descent):

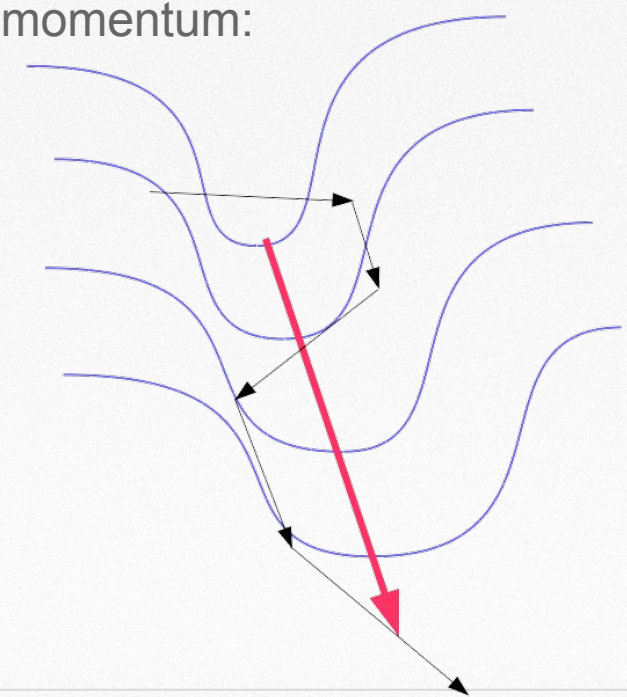
$$v_{k+1} = \eta_k v_k - \alpha_k \nabla h(\theta_k + \eta_k v_k)$$
$$\theta_{k+1} = \theta_k + v_{k+1}$$

Failure case revisited

Gradient descent:



GD w/ momentum:



Comparing the variants of GD w/ momentum

- Nesterov's version:
  - has stronger theoretical guarantees (in the non-stochastic case)
  - exhibits better properties on *certain* real and synthetic example problems
  - *however*, performs almost the same in practice many practical problems

- Differences are bigger when α is large

- Nesterov's version becomes equivalent to standard version as $\alpha \rightarrow 0$

Convergence theory

1st-order methods and lower bounds

- A **first-order method** is defined as one where each update is given by a linear combination of the gradients at previous iterates, i.e.:

$$\theta_{k+1} - \theta_k = d \in \text{Span}\{\nabla h(\theta_0), \nabla h(\theta_1), \ldots, \nabla h(\theta_k)\}$$

- This definition includes:
  - gradient descent *with and without* and momentum
  - more complex methods like Conjugate Gradients (CG)
- Does not include:
  - Any method that multiplies the gradient by some non-trivial matrix (e.g. 2nd-order methods)

Convergence theory

1st-order methods and lower bounds (cont.)

- The following objective function is Lipschitz smooth and strongly convex:

$$h(\theta) = \frac{L-\mu}{8}\left([\theta]_1^2 + \sum_{i=1}^{\infty}([\theta]_{i+1} - [\theta]_i)^2 - 2[\theta]_1\right) + \frac{\mu}{2}\|\theta\|^2$$

- And any first-order method applied to it satisfies the *upper bound*:

$$h(\theta_k) - h(\theta^*) \geq \frac{\mu}{2}\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^{2k}\|\theta_0 - \theta^*\|^2$$

Number of iterations to achieve $h(\theta_k) - h(\theta^*) \leq \epsilon$: $k \in \Omega\left(\sqrt{\kappa}\log\frac{1}{\epsilon}\right)$

Convergence theory

Upper bounds for Nesterov's variant

If $h(\theta)$ is Lipschitz smooth and (locally) strongly convex, θ^* is the (local) minimizer, and α_k and η_k are carefully chosen, then gradient descent w/ Nesterov's momentum satisfies the bound:

$$h(\theta_k) - h(\theta^*) \leq L \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa}} \right)^k \|\theta_0 - \theta^*\|^2$$

where, $\kappa = \dfrac{L}{\mu}$ is a "condition number" = ratio of highest curvature to lowest curvature.

Number of iterations to achieve $h(\theta_k) - h(\theta^*) \leq \epsilon$: $k \in \mathcal{O}\left(\sqrt{\kappa} \log \dfrac{1}{\epsilon} \right)$

Convergence Theory

Comparison of iteration counts

To achieve $h(\theta_k) - h(\theta^*) \leq \epsilon$ the number of iterations k satisfies:

- (Worst-case) lower bound for 1st-order methods: $k \in \Omega\left(\sqrt{\kappa}\log\frac{1}{\epsilon}\right)$
- Upper bound for gradient descent: $k \in \mathcal{O}\left(\kappa\log\frac{1}{\epsilon}\right)$
- Upper bound for GD w/ Nesterov's momentum: $k \in \mathcal{O}\left(\sqrt{\kappa}\log\frac{1}{\epsilon}\right)$

2nd-order methods

Formulation

- Approximate $h(\theta)$ by its 2nd-order Taylor series around current θ :

$$h(\theta + d) \approx h(\theta) + \nabla h(\theta)^\top d + \frac{1}{2} d^\top H(\theta) d$$

- Minimize this local approximation to compute update:

$$-H(\theta)^{-1} \nabla h(\theta) = \arg\min_d \left(h(\theta) + \nabla h(\theta)^\top d + \frac{1}{2} d^\top H(\theta) d \right)$$

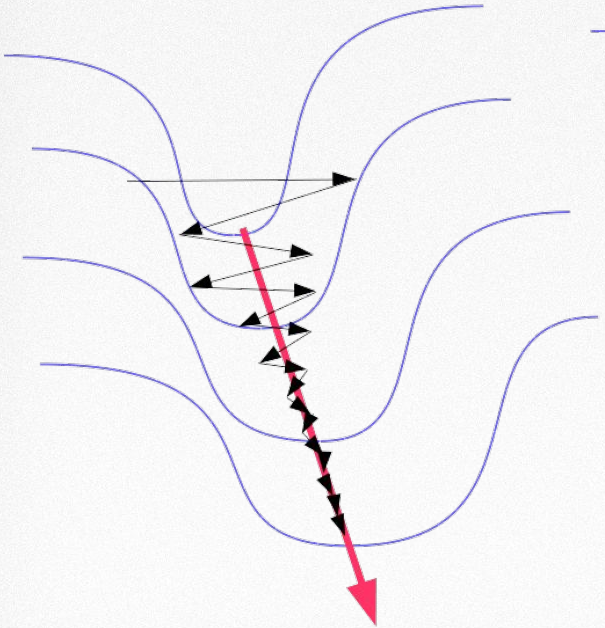
- Update current iterate:

$$\theta_{k+1} = \theta_k - H(\theta)^{-1} \nabla h(\theta_k)$$

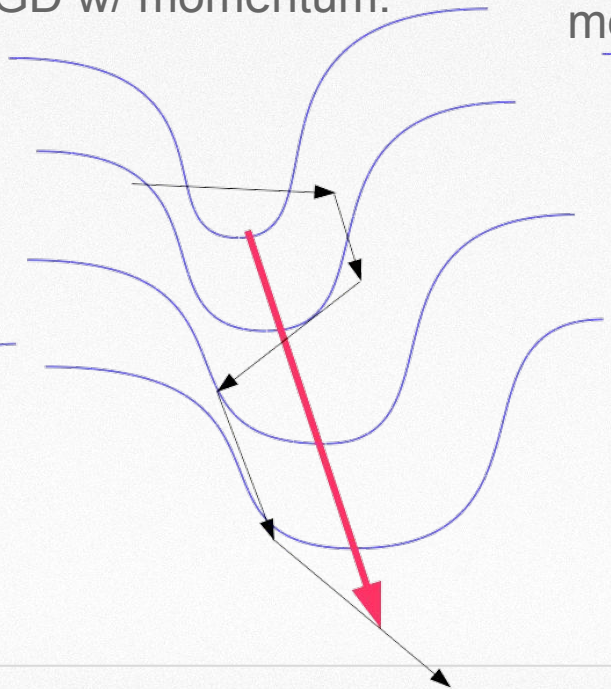
Failure case revisited (again)

2nd-order methods help even more

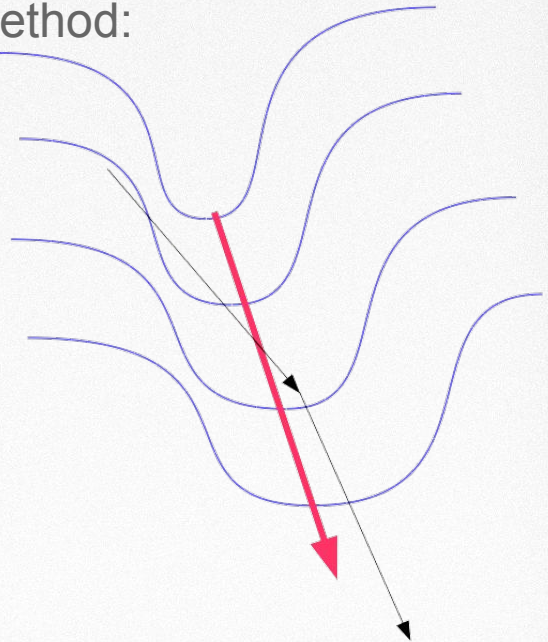
Gradient descent:



GD w/ momentum:



Ideal 2nd-order method:



Problems with naive 2nd-order methods

Breakdown of local approximation

- Approximation is only trustworthy in a local region around current θ

- Unlike gradient descent, which implicitly approximates $LI \approx H(\theta)$ (recall: L upper-bounds the global curvature), the real $H(\theta)$ may underestimate curvature along some directions as we move away from current θ (and curvature may even be *negative!*)

- *Solution:* Constrain update d to lie in some local region R around 0 where approximation remains a good one

$$\arg\min_{d \in R} \left(h(\theta) + \nabla h(\theta)^\top d + \frac{1}{2} d^\top H(\theta) d \right)$$

Trust-regions and Tikhonov regularization/damping

- If we take $R = \{d : \|d\|_2 \leq r\}$ then computing

$$\arg\min_{d \in R} \left(h(\theta) + \nabla h(\theta)^\top d + \frac{1}{2} d^\top H(\theta) d \right)$$

  is often equivalent to computing

$$-(H(\theta) + \lambda I)^{-1} \nabla h(\theta) = \arg\min_{d} \left(h(\theta) + \nabla h(\theta)^\top d + \frac{1}{2} d^\top (H(\theta) + \lambda I) d \right)$$

  for some λ.

- λ is a complicated function of r, but fortunately we can just work with λ directly. There are effective heuristics for adapting lambda such as the "Levenberg-Marquardt" method.

Alternative curvature matrices

Another solution to the model trust problem:

- In place of the Hessian we can use a matrix with more forgiving properties that tends to upper-bound the curvature over larger regions

  (LI is a poor choice because it says “all directions have equal curvature”)

- Very important effective technique in practice if used alongside previously discussed trust-region / regularization / damping techniques

- Some important examples
  - Generalized Gauss-Newton matrix
  - Fisher information matrix
  - Empirical Fisher information matrix

Generalized Gauss-Newton

Definition

- To use the GGN we must assume that

$$h(\theta) = \sum_i h_i(\theta) = \sum_i \ell(y_i, f(x_i, \theta))$$

where

$\ell(y, z)$ is a loss that is convex in z, and

$f(x, \theta)$ is some high-dimensional function (e.g. neural network w/ input x)

- The GGN is given by

$$G = \sum_i J_i^\top H_i J_i$$

where J_i is Jacobian of $f(x_i, \theta)$ w.r.t. θ

and H_i is the Hessian of $\ell(y_i, z_i)$

w.r.t. $z_i = f(x_i, \theta)$

Generalized Gauss-Newton

Derivations and relationship to Fisher

- G is equal to the Hessian of $h(\theta)$ if we replace each $f(x_i, \theta)$ with its local 1st-order approximation centered at current θ:

$$f(x_i, \theta') \approx f(\theta) + J_i(\theta' - \theta)$$

- When $\ell(y, z) = \|y - z\|^2/2$ we have $H_i = I$ and so $\mathrm{G} = \sum_i J_i^\top J_i$
  - this is the matrix used in the well-known Gauss-Newton approach for optimizing nonlinear least squares

- When $\ell(y, z) = -\log p(y|z)$ for a "natural" conditional density $p(y|z)$, G becomes equivalent to Fisher information matrix associated with $p(y|f(x, \theta))$
  - In this case $G^{-1}\nabla h(\theta)$ is equal to the well-known "natural gradient"

The GGN matrix has the following nice properties:

- it always PSD (i.e. models the curvature in all directions as non-negative)
- it is usually more conservative than the Hessian, but won't always be larger in all directions
- an optimizer using $d = -\alpha G^{-1} \nabla h(\theta)$ as its update will be invariant to any smooth reparameterization
- *and most importantly…* works much better than the Hessian in practice for neural networks

Updates computed using the GGN make 100s-1000s times more progress than gradient updates. Unfortunately there is no know way to efficiently compute such updates exactly in high dimensions…

More problems with naive 2nd-order methods

High-dimensional objectives

- For neural networks, $\theta \in \mathbb{R}^n$ can have 10s of millions of dimensions
- We simply cannot compute and store an $n \times n$ matrix for such an n, let alone invert it! ($\mathcal{O}(n^3)$)
- Thus we must approximate the curvature matrix using one of a number of techniques that simplify its structure to allow for efficient
  - computation,
  - storage,
  - and inversion

Curvature matrix approximations

Diagonal approximations:

- Approximate curvature matrix B by its own diagonal: $\hat{B} = \text{diag}(B)$
- Storage cost: $\mathcal{O}(n)$
- Cost to apply inverse (i.e. compute $\hat{B}^{-1}v$): $\mathcal{O}(n)$
- Can be slightly tricky to compute $\hat{B}$ for certain B's, although reasonably efficient estimation methods are available (e.g. "Curvature Propagation")
- Will only be reasonably accurate if eigenvectors of B are closely aligned with the coordinate axes
- A popular choice for B is the "empirical Fisher", which is defined by

$$\sum_i \nabla h_i(\theta) \nabla h_i(\theta)^\top$$

  Several popular diagonal methods use this choice, including "RMS-prop" and "Adam", because the alg to compute the diagonal is simple

Curvature matrix approximations

Low-rank approximations

- Approximate $\hat{B} \approx B$ (or $\widehat{B^{-1}} \approx B^{-1}$) as diagonal + rank-$r$ corrections:

$$\sum_{j=1}^{r} u_j u_j^\top + \mathrm{diag}(u_0)$$

- Moderately easy to store: $\mathcal{O}(rn)$
- Moderately easy to apply inverse: $\mathcal{O}(rn)$
- Moderately easy to compute approx: Usually $\mathcal{O}(rn)$
- Less effective if real B has many important eigenvectors with large eigenvalues
- Most well-known example is L-BFGS

Curvature matrix approximations

Block-diagonal approximations:

- Take $\hat{B}$ to be block-diagonal of B (block size: $b \times b$)
- For neural nets, blocks could correspond to:
  - weights on connections going into a given unit
  - weights on connections going out of a given unit
  - all the weights for a given layer
- Storage cost: $\mathcal{O}(bn)$
- Cost to apply inverse: $\mathcal{O}(b^2 n)$ (just invert each diagonal block)
- Similar difficulty to computing diagonal (and similar methods apply)
- Can only be realistically applied for small block size b
- Well-known example developed for neural nets: TONGA

Curvature matrix approximations

Kronecker-product approximations:

- Block-diagonal approximation of GGN/Fisher where blocks correspond to layers
- Each block is additionally approximated as a Kronecker product two much smaller matrices:

$$A \otimes C = \begin{bmatrix} [A]_{1,1}C & \cdots & [A]_{1,k}C \\ \vdots & \ddots & \vdots \\ [A]_{k,1}C & \cdots & [A]_{k,k}C \end{bmatrix}$$

- Derived by treating unit activations and back-propagated errors as uncorrelated when computing Fisher (= covariance of gradients)
- Storage and computation cost : $\mathcal{O}(n)$
- Cost to apply inverse: $\mathcal{O}(b^{0.5}n)$ (uses $(A \otimes C)^{-1} = A^{-1} \otimes C^{-1}$)
- Current state-of-the-art for neural network optimizers

Stochastic Optimization

Motivation

- Typical objectives in machine learning are an average over training cases of case-specific losses:

$$h(\theta) = \frac{1}{m} \sum_{i=1}^{m} h_i(\theta)$$

- m can be **very** big and so computing the gradient is extremely expensive

$$\nabla h(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla h_i(\theta)$$

Mini-batching

- Fortunately there is often significant statistical overlap between $h_i(\theta)$'s
- Especially early in optimization, when "coarse" features of the data are still being learned, many $\nabla h_i(\theta)$' s will point in roughly the same direction
- Idea: randomly sub-sample a "mini-batch" of training cases $S \subset \{1, 2, ..., m\}$ of size $b \ll m$ and compute the mini-batch gradient:

$$\widetilde{\nabla} h(\theta) = \frac{1}{b} \sum_{i \in S} \nabla h_i(\theta)$$

Stochastic gradient descent

- In stochastic gradient descent (SGD) we replace $\nabla h(\theta)$ with the mini-batch version $\widetilde{\nabla} h(\theta)$ and then compute update as usual:

$$\theta_{k+1} = \theta_k - \alpha_k \widetilde{\nabla} h(\theta_k)$$

- To ensure convergence we need to do one of several things
  - Use a decaying step-size schedule satisfying:

    $\sum_{k=1}^{\infty} \alpha_k^2 < \infty \quad \sum_{k=1}^{\infty} \alpha_k = \infty$ e.g. $\alpha_k = 1/k$
  - Use Polyak averaging:

    $\bar{\theta}_k = \frac{1}{k+1} \sum_{i=0}^{k} \theta_i$ or in practice: $\bar{\theta}_k = (1-\beta)\theta_k + \beta \bar{\theta}_{k-1}$
  - More recently: use variance reduction methods like SAG and SVRG

Stochastic 2nd-order and momentum methods

- For 2nd-order methods also need to compute curvature matrix B without going over whole training set
  - But just computing B on current mini-batch is often not good enough
  - Solution is often to use an exponentially decayed average over time of mini-batch computed B 's (similar to Polyak averaging)
  - This works pretty well, although "staleness" can be a problem

- Momentum can be easily applied to SGD and helps in practice. However:
  - extra care must to be taken with the parameters α and η
  - common practice is to stop using (or lower decay param η) as optimizer gets close to local min

Some convergence theory

- One way to formalize stochastic methods is to treat the stochastic gradient as the random variable

$$\widetilde{\nabla} h(\theta) = \nabla h(\theta) + \varepsilon$$

where ε is some 0-mean noise variable. Often $\varepsilon \sim N(0, \Sigma)$

Note that $E[\widetilde{\nabla} h(\theta)] = \nabla h(\theta)$

- For strongly-convex quadratic objectives stochastic SGD and basic 2nd-order methods, $E[\theta_k]$ behaves the same as the non-stochastic version of the iterate θ_k

Some convergence theory (cont.)

- The theory says:
  - There is no ***asymptotic*** advantage to using 2nd-order methods or momentum over plain SGD w/ *Polyak averaging*
  - Actually, SGD w/ *Polyak averaging* is ***asymptotically optimal*** among any system that tries to estimate parameters of a statistical model by minimizing the loss over bk training cases. The asymptotic rate is:

$$E[h(\theta_k)] - h(\theta^*) \in \mathcal{O}\left(\frac{1}{k}\,\mathrm{tr}\left(H(\theta^*)^{-1}\Sigma\right)\right)$$

  - *However*, ***pre-asymptotically*** there can still be an advantage to using 2nd-order updates and/or momentum
  - Because we care more about pre-asymptotic performance in practice, 2nd-order and momentum methods are still very useful

References/sources

Solid introductory texts:

- *Numerical Optimization* -- by Nocedal & Wright
- *Introductory Lectures on Convex Optimization: A Basic Course* -- by Nesterov

Some possibly relevant papers:

- *The Importance of Initialization and Momentum in Deep Learning* -- by Sutskever et al.
- *New insights and perspectives on the natural gradient method* -- Martens

THANK YOU

Credits

Additional Credits