Stochastic Gradient Descent and Variants

Zhong Hui

University of Science and Technology

zhonghui.net@gmail.com

August 1, 2017

Overview

Outline

Challenges

Both statistical estimation and machine learning consider the problem of minimizing an objective function that has the form of a sum:

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

- Large-scale data, in other words, n is very large. In bigdata times, there are lots of machine learning methods suffered from large-scale data.
- High-dimensions parameters, x's dimensions is very high. Deep neural network are widely used in many problems, it's have huge amounts of parameters

Why Stochastic Methods?

- **Intuitive Motivation** On an intuitive level, SG employs information more efficiently than a batch method.
- **Practical Motivation** The intuitive benefits just described have been observed repeatedly in practice, where one often finds very real advantages of SG in many applications.
- **Theoretical Motivation** One can also cite theoretical arguments for a preference of SG over a batch approach.

Outline

Gradient descent

The negative gradient direction is the most fastest fall direction.

$$\nabla f(x) = g_k$$

The simplest form of update is to change the parameters along the negative gradient direction.

$$x_{k+1} = x_k - \lambda \nabla f(x)$$

where learning rate λ is a hyperparameter - a fixed constant. When evaluated on the full dataset, and when the **learning rate is low enough**, this is guaranteed to make non-negative progress on the loss function.

Annealing the learning rate

Step decay: Reduce the learning rate by some factor every few epochs. Typical values might be reducing the learning rate by a half every 5 epochs, or by 0.1 every 20 epochs. One heuristic you may see in practice is to watch the validation error while training with a fixed learning rate, and reduce the learning rate by a constant (e.g. 0.5) whenever the validation error stops improving.

Exponential decay has the mathematical form $\alpha = \alpha_0 e^{-kt}$, where α_0, k are hyperparameters and t is the iteration number (but you can also use units of epochs).

1/t decay has the mathematical form $\alpha = \alpha_0/(1 + kt)$ where α_0, k are hyperparameters and t is the iteration number.

Momentum update

Momentum update is another approach that almost always enjoys better converge rates on deep networks. This update can be motivated from a physical perspective of the optimization problem.

$$v_{k+1} = \mu * v_k - \lambda * \nabla f(x)$$
$$x_{k+1} = x_k + v_{k+1}$$

$$SGD : gradient \rightarrow position$$
$$Momentum : gradient \rightarrow velocity \rightarrow position$$

Note that this is different from the SGD update shown above, where the gradient directly integrates the position. Instead, the physics view suggests an update in which the gradient only directly influences the velocity, which in turn has an effect on the position.

With Momentum update, the parameter vector will build up velocity in any direction that has consistent gradient.

Nesterov Momentum (Nesterov and Nemirovskii, 1994)

Nesterov Momentum is a slightly different version of the momentum update that has recently been gaining popularity.

$$\begin{aligned} x_{ahead} &= x_k + \mu * v_k \\ v_{k+1} &= \mu * v_k - \lambda * \nabla f(x_{ahead}) \\ x_{k+1} &= x_k + v_{k+1} \end{aligned}$$

It enjoys stronger theoretical converge guarantees for convex functions $O(\frac{1}{k^2})$, and in practice it also consistently works **slightly better than standard momentum**.

Nesterov Momentum

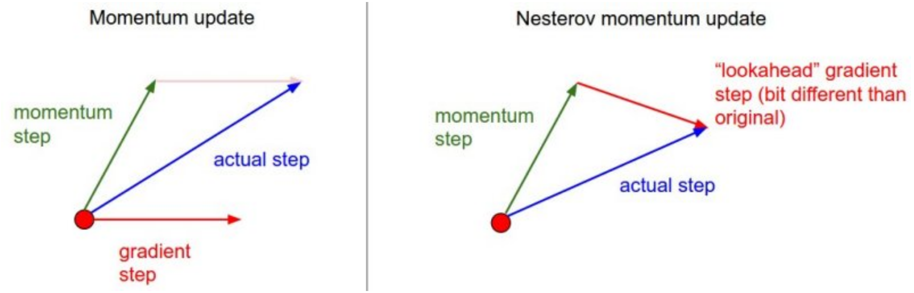


Figure: Nesterov momentum. Instead of evaluating gradient at the current position (red circle), we know that our momentum is about to carry us to the tip of the green arrow. With Nesterov momentum we therefore instead evaluate the gradient at this "looked-ahead" position.

Outline

Newton's method

Consider about the second order Taylor expansion of $f(x)$ at x_k point

$$\begin{aligned} f(x) &= f(x_k) + g_k^T(x - x_k) + \frac{1}{2}(x - x_k)^T H(x_k)(x - x_k) \\ \nabla f(x) &= g_k + H(x_k)(x - x_k) \end{aligned}$$

let $\nabla f(x) = 0$, so $x - x_k = -H(x_k)^{-1}g(x)$. so we get the unpdate form of Newton's method.

$$x_{k+1} = x_k - H(x_k)^{-1}\nabla f(x_k)$$

Quasi-Newton's method (Shanno, 1970)

The derivative is

$$\nabla f(x) = g_k + H(x_k)(x - x_k)$$

if we let $x = x_{k+1}$, then $\nabla f(x) = g_{k+1}$ we get

$$g_{k+1} - g_k = H_k(x_{k+1} - x_k)$$

let $y_k = g_{k+1} - g_k$, $\delta_k = x_{k+1} - x_k$, $y_k = H_k\delta_k$.
DFP $G_{k+1} \approx H_k^{-1}$, BFGS $B_{k+1} \approx H_k$

$$G_{k+1}y_k = \delta_k$$
$$B_{k+1}\delta_k = y_k$$

Newton's method (Atkinson, 2008)

Intuitively, the Hessian describes the **local curvature** of the loss function, which allows us to perform a more efficient update.
Note, crucially, the **absence of any learning rate** hyperparameters in the update formula, which the proponents of these methods cite this as a large advantage over first-order methods.

$$x_{k+1} = x_k - H(x_k)^{-1} \nabla f(x_k)$$

In practice, it is currently **not common** to see L-BFGS or similar **second-order methods** applied to large-scale Optimization. Instead, SGD variants based on (Nesterov's) momentum are more standard because they are simpler and scale more easily.

Outline

AdaGrad Full (Duchi et al., 2011)

Making SGD adaptive to the data geometry

$$x_{k+1} = \prod_{\mathcal{W}}^{G_t^{\frac{1}{2}}} (x_k - \lambda G_t^{-\frac{1}{2}} g_t)$$
$$= argmin_{x \in \mathcal{W}} \left\| x - (x_k - \lambda G_t^{-\frac{1}{2}} g_t) \right\|_{G_t^{\frac{1}{2}}}^2$$

where $G_t := \sum_{\tau=0}^{t} g_\tau g_\tau^T$ is a covariance matrix estimated from historical subgradients for 2nd-order correction

AdaGrad Approximation

- Similar in spirit to quasi-Newton SGD (Bordes et al., 2009)

$$x_{k+1} = \prod_{\mathcal{W}}^{\tilde{H}_t} (x_k - \lambda \tilde{H}_t^{-1} g_t)$$

- But is more widely applicable, e.g., to nonsmooth problems

Because of $G_t^{\frac{1}{2}}$ is computationally impractical for large dimensions. In pratice, we usually replace G_t with its diagonal:

$$\begin{aligned} v_{k+1} =& v_k + [\nabla f(x)]^2 \\ x_{k+1} =& x_k - \lambda \frac{\nabla f(x)}{\sqrt{v_{k+1}} + \epsilon} \end{aligned}$$

AdaGrad - Takeaways

AdaGrad is a variant of SGD which

- Adaptively adjusts to data geometry
- Adaptively determines learning rates for different dimensions

Advantages of AdaGrad

- Easy to implement and tends to work well in practice. Not sensitive to the initial learning rate.
- Generic, backed up by nice theoretical guarantees $Regret(T)/T : O(\frac{1}{\sqrt{T}})$.

AdaDelta

A recent method by Schaul (Schaul et al., 2013) incorporating the diagonal Hessian with ADAGRAD-like terms has been introduced to alleviate the need for hand specified learning rates. This method uses the following update rule:

$$\Delta x_t = -\frac{1}{|\text{diag}(H_t)|}\frac{E[g_{t-w:t}]^2}{E[g_{t-w:t}^2]}\, g_t$$

where $E[g_{t-w:t}]$ is the expected value of the previous w gradients and $E[g_{t-w:t}^2]$ is the expected value of squared gradients over the same window w. (see Schaul et al. (2013) for more details).

AdaDelta (Zeiler, 2012)

When considering the parameter updates, Δx, being applied to x, the units should match. That is, if the parameter had some hypothetical units, the changes to the parameter should be changes in those units as well. The units in SGD and Momentum relate to the gradient, not the parameter x :

$$\text{units of } \Delta x \propto \text{units of } g \propto \frac{\partial f}{\partial x} \propto \frac{1}{\text{units of } \ x}$$

assuming the cost function, f, is unitless. In contrast, second order methods such as Newton's method that use Hessian information or an approximation to the Hessian do have the correct units for the parameter updates:

$$\Delta x \propto H^{-1} g \propto \frac{\frac{\partial f}{\partial x}}{\frac{\partial^2 f}{\partial x^2}} \propto \text{units of } x$$

AdaDelta Update Form

The method uses only first order information.The negative gradient direction for the current iteration $-g_t$ is always followed as in SGD.

$$\begin{aligned} v_k =& \beta * v_{k-1} + (1-\beta) * [\nabla f(x_k)]^2 \\ \Delta x_k =& -\frac{\sqrt{[\Delta x^2]_{k-1}}}{\sqrt{v_k}} * \nabla f(x_k) \\ [\Delta x^2]_k =& \beta * [\Delta x^2]_{k-1} + (1-\beta) * \Delta x_k^2 \\ x_{k+1} =& x_k + \Delta x_k \end{aligned}$$

The numerator acts as an acceleration term. The denominator is related to ADAGRAD in that the squared gradient information per-dimension helps to even out the progress made in each dimension.
the method relates to Schaul's, but instead costs only one gradient computation per iteration by leveraging information from past updates.

RMSprop (Tieleman and Hinton, 2012)

RMSProp (for Root Mean Square Propagation). Similar method rprop(resilient backpropagation), see (Riedmiller and Braun, 1993) for detail.
The RMSProp update adjusts the Adagrad method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate. In particular, it uses a moving average of squared gradients instead, giving:

$$\begin{aligned} v_{k+1} =& \beta * v_k + (1-\beta) * [\nabla f(x)]^2 \\ x_{k+1} =& x_k - \lambda \frac{\nabla f(x)}{\sqrt{v_{k+1}} + \epsilon} \end{aligned}$$

Here, β is a hyperparameter. Notice that the x update is identical to Adagrad, but the v_k variable is a "leaky".
Hence, RMSProp still modulates the learning rate of each weight based on the magnitudes of its gradients, which has a beneficial equalizing effect, but unlike Adagrad the updates do not get monotonically smaller.

RMSprop

Since we add momentum on $[\nabla f(x)]^2$, why not also add momentum on gradient?

The answer is Yes!

$$\begin{aligned} v_{k+1} =& \beta * v_k + (1-\beta) * [\nabla f(x)]^2 \\ x_{k+1} =& x_k - \lambda \frac{\nabla f(x)}{\sqrt{v_{k+1}} + \epsilon} \end{aligned}$$

Adam (Kingma and Ba, 2014)

The method combines the advantages of two recently popular optimization methods:
the ability of AdaGrad to deal with sparse gradients, and the ability of RMSProp to deal with non-stationary objectives. The method is straightforward to implement and requires little memory.

$$\begin{aligned} m_{k+1} =& \beta_1 * m_k + (1-\beta_1) * \nabla f(x) \\ v_{k+1} =& \beta_2 * v_k + (1-\beta_2) * [\nabla f(x)]^2 \\ x_{k+1} =& x_k - \lambda * \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \frac{m_{k+1}}{\sqrt{v_{k+1}}+\epsilon} \end{aligned}$$

Adam bias correction

$$v_t = (1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}\cdot g_i^2 \tag{1}$$

We wish to know how $\mathbb{E}[v_t]$, the expected value of the exponential moving average at timestep t, relates to the true second moment $\mathbb{E}[g_t^2]$, so we can correct for the discrepancy between the two. Taking expectations of the left-hand and right-hand sides of eq. (1):

$$\begin{aligned}\mathbb{E}[v_t] &= \mathbb{E}\left[(1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}\cdot g_i^2\right]\\ &= \mathbb{E}[g_t^2]\cdot(1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}+\zeta\\ &= \mathbb{E}[g_t^2]\cdot(1-\beta_2^t)+\zeta\end{aligned}$$

Extensions: AdaMax

we can generalize the L^2 norm based update rule to a L^p norm based update rule.

$$\begin{aligned} v_t &= \beta_2^p v_{t-1} + (1-\beta_2^p)|g_t|^p \\ &= (1-\beta_2^p)\sum_{i=1}^{t} \beta_2^{p(t-i)} \cdot |g_i|^p \end{aligned}$$

Note that the decay term is here equivalently parameterised as β_2^p instead of β_2. Now let $p \to \infty$, and define $u_t = \lim_{p\to\infty}(v_t)^{1/p}$, then:

$$\begin{aligned} u_t = \lim_{p\to\infty}(v_t)^{1/p} &= \lim_{p\to\infty}\left((1-\beta_2^p)\sum_{i=1}^{t} \beta_2^{p(t-i)} \cdot |g_i|^p\right)^{1/p} \\ &= \max\left(\beta_2^{t-1}|g_1|, \beta_2^{t-2}|g_2|, \ldots, \beta_2|g_{t-1}|, |g_t|\right) \end{aligned}$$

Which corresponds to the remarkably simple recursive formula:

$$u_t = \max(\beta_2 \cdot v_{t-1}, |g_t|)$$

Adam Compare

RMSProp with momentum generates its parameter updates using a momentum on the rescaled gradient, whereas Adam updates are directly estimated using a running average of first and second moment of the gradient.
RMSProp also lacks a bias-correction term; this matters most in case of a value of β_2 close to 1 (required in case of sparse gradients), since in that case not correcting the bias leads to very large stepsizes and often divergence.

AdaGrad: Note that if we choose β_2 to be infinitesimally close to 1 from below, then $\lim_{\beta_2 \to 1} \tilde{v} = t^{-1} \cdot \sum_{i=1}^{t} g_t^2$. AdaGrad corresponds to a version of Adam with $\beta_1 = 0$, infinitesimal $(1 - \beta_2)$ and a replacement of α by an annealed version $\alpha_t = \alpha \cdot t^{-1/2}$.
without bias correction, like in RMSProp, a β_2 infinitesimally close to 1 would lead to infinitely large bias, and infinitely large parameter updates.

Adam Tricks

When $(1-\beta_1) = \sqrt{1-\beta_2}$ we have that $|\tilde{m}/\sqrt{\tilde{v}}| < 1$ therefore $|\Delta_t| < \alpha$. In more common scenarios, we will have that $\tilde{m}/\sqrt{\tilde{v}} \approx \pm 1$ since $|\mathbb{E}[g]/\sqrt{\mathbb{E}[g^2]}| \leq 1$.

So, if we randomly init x in $range(-1, 1)$, we should ensure that every dimension of x can arrive at it's optimal value.

$$run\_times = \frac{1-(-1)}{learn\_rate}$$

Outline

SAGA revisit (Defazio et al., 2014)

Iterations form

$$x^{k+1} = x^k - \alpha \left[f_j'(x^k) - f_j'(\phi_j^k) + \frac{1}{n}\sum_{i=1}^{n} f_i'(\phi_i^k) \right]$$

index j is sampled uniformly from the set $\{1, ..., n\}$. $\phi_j^k = x_{k-1}$, and store $f_j'(\phi_j^k)$ in the table of all $\sum f_i'(\phi_i^k)$ sets.

the same convergence rate as FG, *linear convergence rate* $O(\rho^k)$ for strongly-convex objectives, $O(1/k)$ for convex objectives.

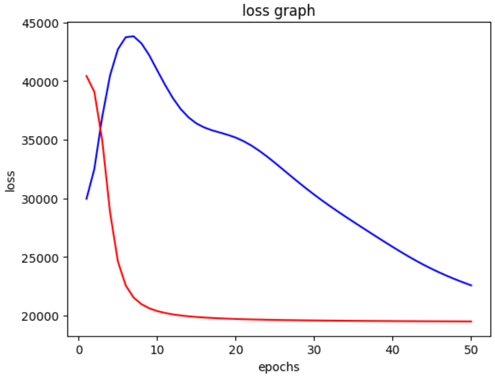
Algorithm

Algorithm 1 Adaptive momentum - sgd sag and saga (x_0, T_{MAX}, n, f_i)

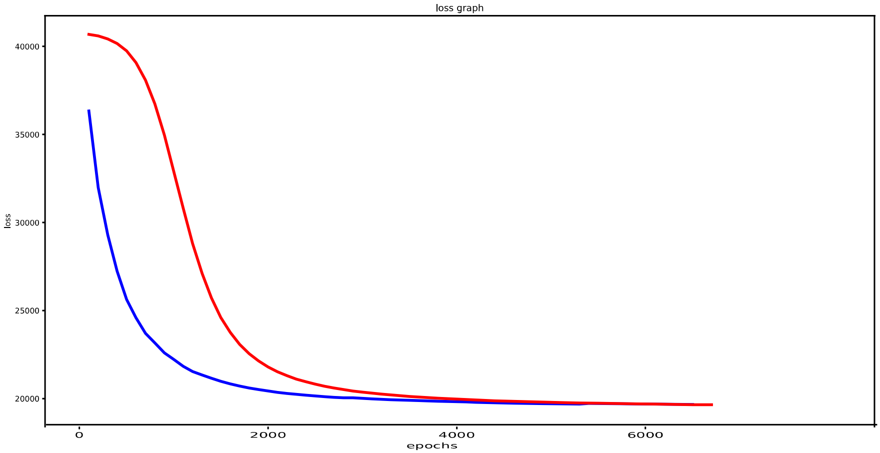
```
**Input:** $x^0 \in \mathbb{R}^d$, $K$, step sizes $\lambda = 0.001$,$\alpha = 0.9$
**Init**
**while** not converge OR $k < T_{MAX}$ **do**
    $k \leftarrow k + 1$
    Random pick $i$ from $1 \to n$ , calculate $g_{ik} \leftarrow \nabla f_i(x_k)$
    $V_k \leftarrow \alpha V_{k-1} + (1-\alpha) g_{ik}^2$
    $m_i \leftarrow g_{ik}$
    Optional Step:
            **SGD:**
                $x_k = x_{k-1} - \lambda \frac{m_i}{\sqrt{V_k} + \epsilon}$
            **SAG:**
                $x_k = x_{k-1} - \lambda \frac{\frac{1}{n}\sum_i^n m_i}{\sqrt{V_k} + \epsilon}$
            **SAGA:**
                $x_k = x_{k-1} - \lambda \frac{m_i - m_{i-1} + \frac{1}{n}\sum_i^n m_i}{\sqrt{V_k} + \epsilon}$
    *(proximal step: $x_k = prox_h^x$)
    store $m_i, V_k$
**end while**

**Output:** $x^k$
```

Why Can't add Momentum on first order gradient?

In SAGA, we use the gradients on all past mini-batchs. if we add momentum on every batch, the effects of momentum exponential decay on past gradeients will diffuse in every batch.



simple experiment result



Outline

Reference

Atkinson, K. E. (2008). *An introduction to numerical analysis*. John Wiley & Sons.

Bordes, A., Bottou, L., and Gallinari, P. (2009). Sgd-qn: Careful quasi-newton stochastic gradient descent. *Journal of Machine Learning Research*, 10(Jul):1737–1754.

Defazio, A., Bach, F., and Lacoste-Julien, S. (2014). Saga: A fast incremental gradient method with support for non-strongly convex composite objectives. In *Advances in Neural Information Processing Systems*, pages 1646–1654.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*.

Nesterov, Y. and Nemirovskii, A. (1994). *Interior-point polynomial algorithms in convex programming*. SIAM.

Riedmiller, M. and Braun, H. (1993). A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE.

Schaul, T., Zhang, S., and LeCun, Y. (2013). No more pesky learning rates. In *International Conference on Machine Learning*, pages 343–351.

Shanno, D. F. (1970). Conditioning of quasi-newton methods for function minimization. *Mathematics of computation*, 24(111):647–656.

Tieleman, T. and Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31.

Demo

Thank You