A Brief (and Comprehensive) Guide to Stochastic Gradient Descent Algorithms

10/03/2017
ARTIFICIAL INTELLIGENCE, DEEP LEARNING, GENERIC, MACHINE LEARNING, MACHINE LEARNING ALGORITHMS ADDENDA, NEURAL NETWORKS
NO COMMENTS

Stochastic Gradient Descent (SGD) is a very powerful technique, currently employed to optimize all deep learning models. However, the *vanilla* algorithm has many limitations, in particular when the system is ill-conditioned and could never find the global minimum. In this post, we're going to analyze how it works and the most important variations that can speed up the convergence in deep models.

First of all, it's necessary to standardize the naming. In some books, the expression "**Stochastic Gradient Descent**" refers to an algorithm which operates on a batch size equal to 1, while "**Mini-batch Gradient Descent**" is adopted when the batch size is greater than 1. In this context, we assume that Stochastic Gradient Descent operates on batch sizes equal or greater than 1. In particular, if we define the loss function for a single sample as:

$$L(\bar{x}_i, \bar{y}_i; \bar{\theta})$$

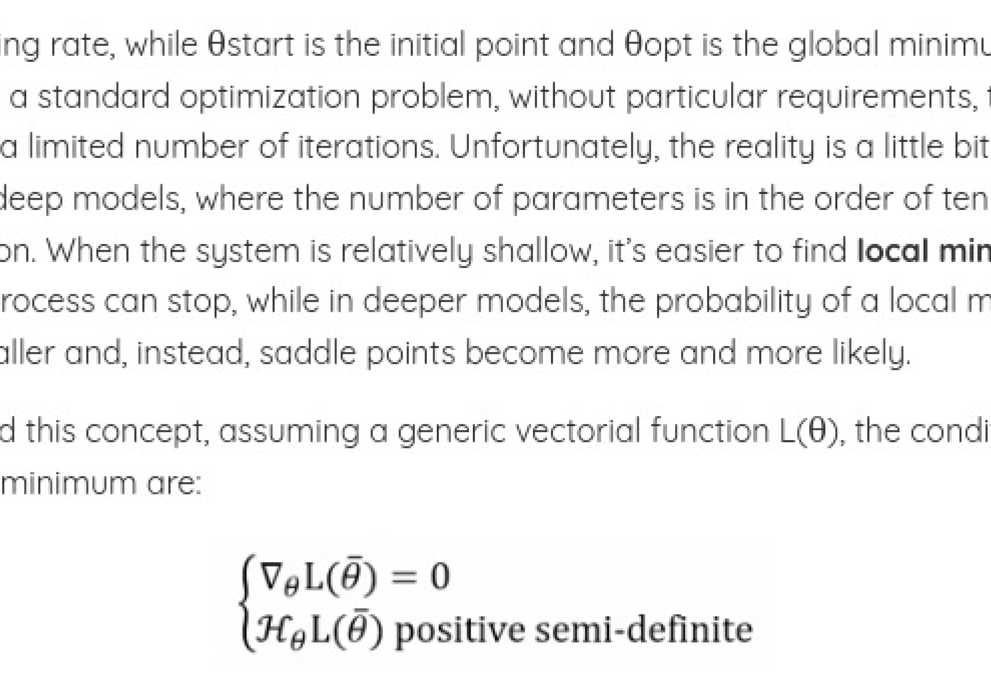
where x is the input sample, y the label(s) and θ is the parameter vector, we can also define the *partial* cost function considering a batch size equal to N:

$$L(\bar{\theta}) = \frac{1}{N}\sum_{i=1}^{N} L\left(\bar{x}_i, \bar{y}_i; \bar{\theta}\right)$$

The *vanilla* Stochastic Gradient Descent algorithm is based on a θ update rule that must move the weights in opposite direction of the gradient of L (the gradient points always toward a maximum):

$$\bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} - \alpha\nabla_\theta\, L(\bar{\theta})$$

This process is represented in the following figure:



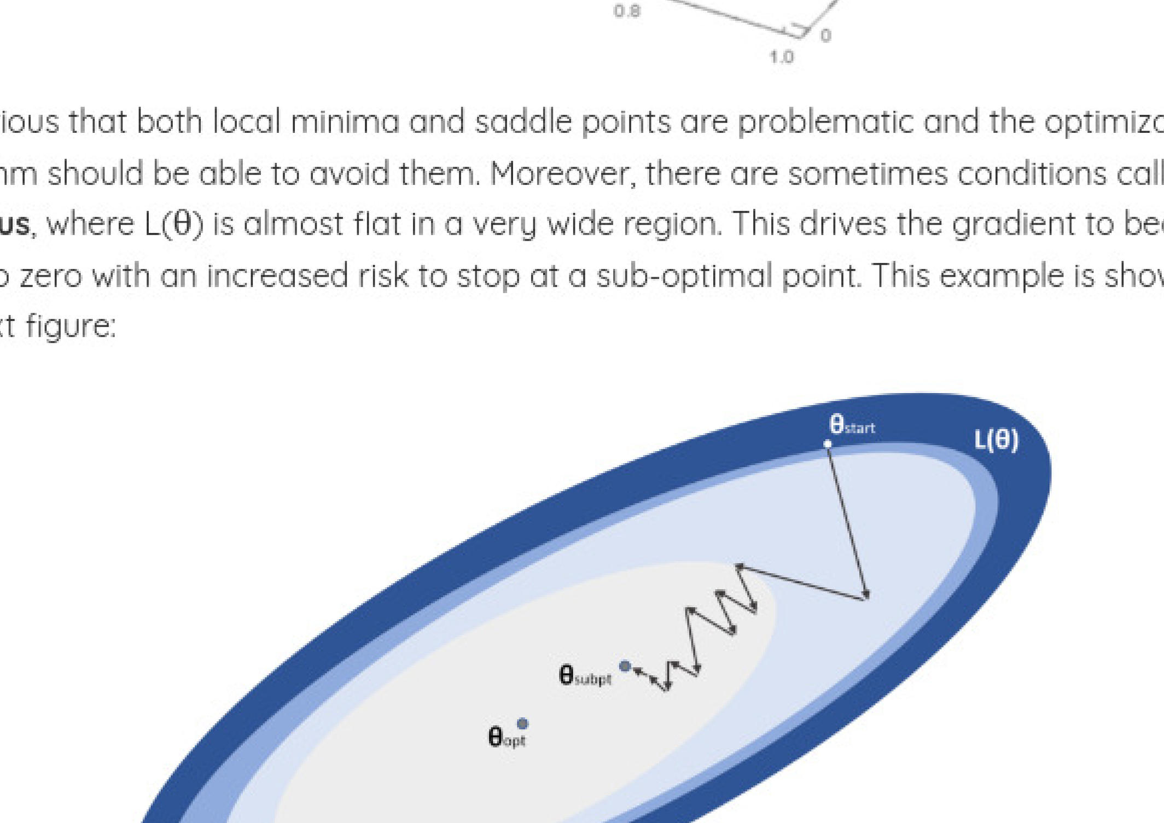
α is the learning rate, while θstart is the initial point and θopt is the global minimum we're looking for. In a standard optimization problem, without particular requirements, the algorithm converges in a limited number of iterations. Unfortunately, the reality is a little bit different, in particular in deep models, where the number of parameters is in the order of ten or one hundred million. When the system is relatively shallow, it's easier to find **local minima** where the training process can stop, while in deeper models, the probability of a local minimum becomes smaller and, instead, saddle points become more and more likely.

To understand this concept, assuming a generic vectorial function L(θ), the conditions for a point to be a minimum are:

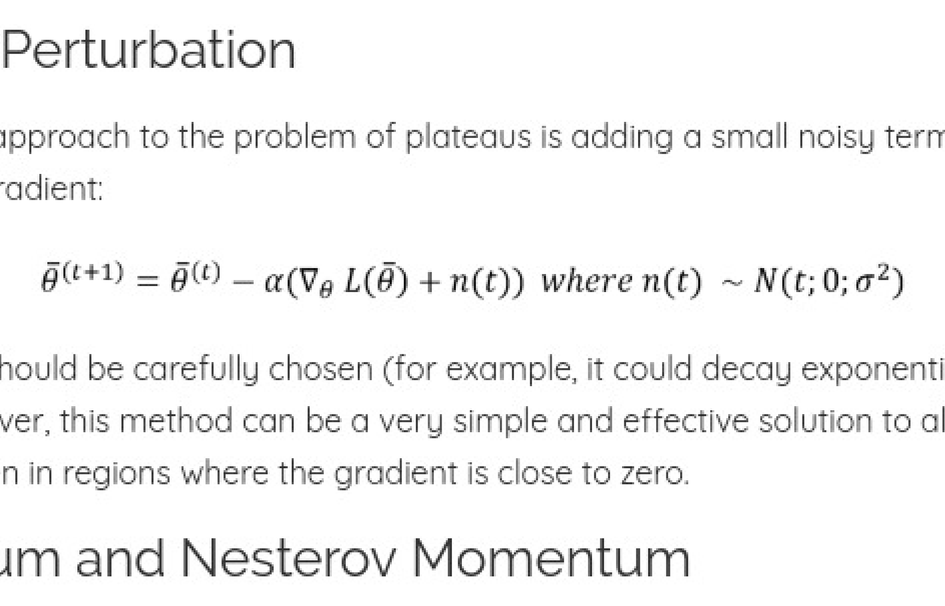
$$\begin{cases} \nabla_\theta \mathrm{L}(\bar{\theta}) = 0 \\ \mathcal{H}_\theta \mathrm{L}(\bar{\theta}) \text{ positive semi-definite} \end{cases}$$

If the Hessian matrix is (m x m) where m is the number of parameters, the second condition is equivalent to say that all m eigenvalues must be non-negative (or that all principal minors composed with the first n rows and n columns must be non-negative).

From a probabilistic viewpoint, P(H positive semi-def.) → 0 when m → ∞, therefore local minima are rare in deep models (this doesn't mean that local minima are impossible, but their relative *weight* is lower and lower in deeper models). If the Hessian matrix has both positive and negative eigenvalues (and the gradient is null), the Hessian is said to be indefinite and the point is called **saddle point**. In this case, the point is maximum considering an orthogonal projection and a minimum for another one. In the following figure, there's an example created with a 3-dimensional cost function:



It's obvious that both local minima and saddle points are problematic and the optimization algorithm should be able to avoid them. Moreover, there are sometimes conditions called **plateaus**, where L(θ) is almost flat in a very wide region. This drives the gradient to become close to zero with an increased risk to stop at a sub-optimal point. This example is shown in the next figure:



For these reasons, now we're going to discuss some common methods to improve the performance of a *vanilla* Stochastic Gradient Descent algorithm.

Gradient Perturbation

A very simple approach to the problem of plateaus is adding a small noisy term (Gaussian noise) to the gradient:

$$\bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} - \alpha(\nabla_\theta\, L(\bar{\theta}) + n(t))\ \ where\ n(t)\ \sim N(t; 0; \sigma^2)$$

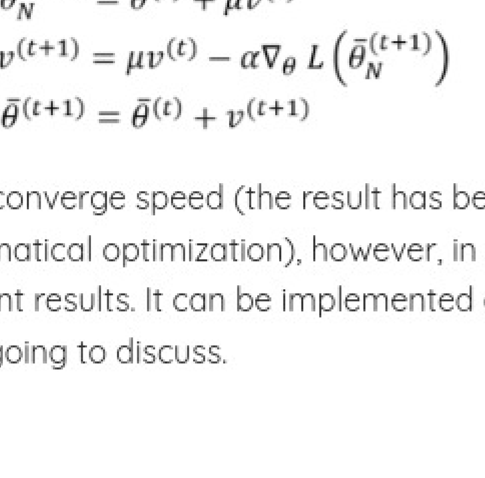
The variance should be carefully chosen (for example, it could decay exponentially during the epochs). However, this method can be a very simple and effective solution to allow a *movement* even in regions where the gradient is close to zero.

Momentum and Nesterov Momentum

The previous approach was quite simple and in many cases, it can difficult to implement. A more robust solution is provided by introducing an exponentially weighted moving average for the gradients. The idea is very intuitive: instead of considering only the current gradient, we can *attach* part of its history to the correction factor, so to avoid an abrupt change when the surface becomes flat. The **Momentum** algorithm is, therefore:

$$\begin{cases} v^{(t+1)} = \mu v^{(t)} - \alpha\nabla_\theta\, L(\bar{\theta}) \\ \bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} + v^{(t+1)} \end{cases}$$

The first equation computes the correction factor considering the weight μ. If μ is small, the previous gradients are soon discarded. If, instead, μ → 1, their effect continues for a longer time. A common value in many application is between 0.75 and 0.99, however, it's important to consider μ as a hyperparameter to adjust in every application. The second term performs the parameter update. In the following figure, there's a vectorial representation of a Momentum step:



A slightly different variation is provided by the **Nesterov Momentum**. The difference with the base algorithm is that we first apply the correction with the current factor v(t) to determine the gradient and then compute v(t+1) and correct the parameters:

$$\begin{cases} \bar{\theta}_N^{(t+1)} = \bar{\theta}^{(t)} + \mu v^{(t)} \\ v^{(t+1)} = \mu v^{(t)} - \alpha\nabla_\theta\, L\left(\bar{\theta}_N^{(t+1)}\right) \\ \bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} + v^{(t+1)} \end{cases}$$

This algorithm can improve the converge speed (the result has been theoretically proven by Nesterov in the scope of mathematical optimization), however, in deep learning contexts, it doesn't seem to produce excellent results. It can be implemented alone or in conjunction with the other algorithms that we're going to discuss.

RMSProp

This algorithm, proposed by G. Hinton, is based on the idea to adapt the correction factor for each parameter, so to increase the effect on slowly-changing parameters and reduce it when their change magnitude is very large. This approach can dramatically improve the performance of a deep network, but it's a little bit more expensive than Momentum because we need to compute a *speed* term for each parameter:

$$v^{(t+1)}(\theta_i) = \mu v^{(t)}(\theta_i) + (1-\mu)\left(\nabla_\theta\, L(\bar{\theta})\right)^2$$

This term computes the exponentially weighted moving average of the gradient squared (element-wise). Just like for Momentum, μ determines how fast the previous speeds are forgotten. The parameter update rule is:

$$\bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} - \frac{\alpha}{\sqrt{v^{(t+1)}(\bar{\theta}) + \delta}}\nabla_\theta\, L(\bar{\theta})$$

α is the learning rate and δ is a small constant (~ 1e-6 ÷ 1e-5) introduced to avoid a division by zero when the speed is null. As it's possible to see, each parameter is updated with a rule that is very similar to the *vanilla* Stochastic Gradient Descent, but the actual learning rate is adjusted per single parameter using the reciprocal of the square root of the relative speed. It's easy to understand that large gradients determine large speeds and, adaptively, the corresponding update is smaller and vice-versa. **RMSProp** is a very powerful and flexible algorithm and it is widely used in Deep Reinforcement Learning, CNN, and RNN-based projects.

Adam

Adam is an adaptive algorithm that could be considered as an extension of **RMSProp**. Instead of considering the only exponentially weighted moving average of the gradient square, it computes also the same value for the gradient itself:

$$\begin{cases} g^{(t+1)}(\theta_i) = \mu_1 g^{(t)}(\theta_i) + (1-\mu_1)\nabla_\theta\, L(\bar{\theta}) \\ v^{(t+1)}(\theta_i) = \mu_2 v^{(t)}(\theta_i) + (1-\mu_2)\left(\nabla_\theta\, L(\bar{\theta})\right)^2 \end{cases}$$

μ1 and μ2 are forgetting factors like in the other algorithms. The authors suggest values greater than 0.9. As both terms are moving estimations of the first and the second moment, they can be biased (see this article for further information). Adam provided a bias correction for both terms:

$$\begin{cases} \hat{g}(\theta_i) = \dfrac{g^{(t+1)}(\theta_i)}{1-\mu_1{}^{(t)}} \\ \hat{v}(\theta_i) = \dfrac{v^{(t+1)}(\theta_i)}{1-\mu_2{}^{(t)}} \end{cases}$$

The parameter update rule becomes:

$$\bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} - \frac{\alpha g^{(t+1)}(\bar{\theta})}{\sqrt{v^{(t+1)}(\bar{\theta}) + \delta}}$$

This rule can be considered as a standard RMSProp one with a momentum term. In fact, the correction term is made up of: (numerator) the moving average of the gradient and (denominator) the adaptive factor to modulate the magnitude of the change so to avoid different changing *amplitudes* for different parameters. The constant δ, like for RMSProp, should be set equal to 1e-6 and it's necessary to improve the numerical stability.

AdaGrad

This is another adaptive algorithm based on the idea to consider the historical sum of the gradient square and set the correction factor of a parameter so to scale its value with the reciprocal of the squared historical sum. The concept is not very different from RMSProp and Adam, but, in this case, we don't use an exponentially weighted moving average, but the whole history. The accumulation step is:

$$g^{(t+1)}(\bar{\theta}) = g^{(t)}(\bar{\theta}) + \left(\nabla_\theta\, L(\bar{\theta})\right)^2$$

While the update step is exactly as RMSProp:

$$\bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} - \frac{\alpha}{\sqrt{g^{(t+1)}(\bar{\theta}) + \delta}}\nabla_\theta\, L(\bar{\theta})$$

AdaGrad shows good performances in many tasks, increasing the convergence speed, but it has a drawback deriving from accumulating the whole squared gradient history. As each term is non-negative, g → ∞ and the correction factor (which is the adaptive learning rate) → 0. Therefore, during the first iterations, AdaGrad can produce significant changes, but at the end of a long training process, the change rate is almost null.

AdaDelta

AdaDelta is algorithm proposed by M. D. Zeiler to solve the problem of AdaGrad. The idea is to consider a limited window instead of accumulating for the whole history. In particular, this result is achieved using an exponentially weighted moving average (like for RMSProp):

$$v^{(t+1)}(\theta_i) = \mu v^{(t)}(\theta_i) + (1-\mu)\left(\nabla_\theta\, L(\bar{\theta})\right)^2$$

A very interesting expedient introduced by AdaDelta is to normalize the parameter updates so to have the same unit of the parameter. In fact, if we consider RMSProp, we have:

$$units\ of\ \Delta\theta \propto \frac{units\ of\ \nabla_\theta\, L(\bar{\theta})}{\sqrt{units\ of\ g^{(t+1)}(\bar{\theta})}} \propto \frac{\frac{1}{units\ of\ \bar{\theta}}}{\sqrt{\frac{1}{(units\ of\ \bar{\theta})^2}}} \propto 1$$

This means that an update is unitless. To avoid this problem, AdaDelta computes the exponentially weighted average of the squared updates and apply this update rule:

$$\begin{cases} u^{(t+1)} = \mu u^{(t)} + (1-\mu)(\Delta\bar{\theta})^2 \\ \bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} - \dfrac{\sqrt{u^{(t)}}}{\sqrt{v^{(t+1)}(\bar{\theta}) + \delta}}\nabla_\theta\, L(\bar{\theta}) \end{cases}$$

This algorithm showed very good performances in many deep learning tasks and it's less expensive than AdaGrad. The best value for the constant δ should be assessed through cross-validation, however, 1e-6 is a good default for many tasks, while μ can be set greater than 0.9 in order to avoid a predominance of old gradients and updates.

Conclusions

Stochastic Gradient Descent is intrinsically a powerful method, however, in non-convex scenarios, its performances can be degraded. We have explored different algorithms (most of them are currently the first choice in deep learning tasks), showing their strengths and weaknesses. Now the question is: which is the best? The answer, unfortunately, doesn't exist. All of them can perform well in some contexts and bad in others. In general, all adaptive methods tend to show similar behaviors, but every problem is a separate universe and the only silver bullet we have is trial and error. I hope the exploration has been clear and any constructive comment or question is welcome!

References:

- Goodfellow I., Bengio Y., Courville A., Deep Learning, The MIT Press
- Duchi J., Hazan E., Singer Y., Adaptive Subgradient Methods for Online Learning and Stochastic Optimization, Journal of Machine Learning Research 12 (2011) 2121-2159
- Zeiler M. D., AdaDelta: An Adaptive Learning Rate Method, arXiv:1212.5701

See also: