Paul Champion - M2A Sorbonne Université

An overview of gradient descent optimization algorithms

Gradient descent algorithm are the most common way to optimize weights and biases in neural networks.

Formally, gradient-descent algorithms aim at minimizing L a function over $\Theta \subset \mathbb{R}^d$

$$\min_{\theta \in \Theta} L(\theta)$$

Assuming Θ convex and J differentiable over Θ, the problem aforementioned is solved in an iterative fashion.

$$\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t)$$

where η is called the learning rate, with positive value.

Gradient descent

In the context of deep-learning algorithms, the loss function $L_{\mathbf{x},\mathbf{y}}(\theta)$ is computed as a function of the parameters θ for a given input vector $\mathbf{x}$ and output $\mathbf{y}$.

For for a given number of iterations (also called **epochs**), the gradient descent algorithm is implemented as follows

1. Initialization with random values of parameters θ_0
2. For a given epoch $n > 1$
   A. Selection of the sample $\mathbf{x}, \mathbf{y}$
   B. Computation of the loss gradient $\nabla_\theta L_{\mathbf{x},\mathbf{y}}(\theta_{n-1})$
   C. Calculation of the parameter at epoch n with the gradient descent formula:
   $\theta_n = \theta_{n-1} - \eta \nabla_\theta L_{\mathbf{x},\mathbf{y}}(\theta_{n-1})$, with η a given constant, called the learning rate

Depending on the considered sample, there are three types of algorithms:

- Stochastic Gradient Descent (SGD): the considered sample $(\mathbf{x}, \mathbf{y})$ is correspond to one sample of the dataset, sampled randomly.
- Batch descent: the considered sample is the whole dataset available.
- Mini-batch descent: the considered sample is a sample of a given size, sampled randomly from the dataset.

The choice of the suitable algorithm for a given application revolves around the trade-off between computation and convergence time. SGD is computed much faster than Batch gradient descent, since there is only one row to consider at each epoch compared to the whole dataset. On the other hand, convergence of the loss to zero is steady with Batch gradient descent whereas it may be erratic with SGD. Minibatch mitigates both downside by taking a given number of sample randomly in the dataset.

Momentum methods

Momentum

SGD descent does not converge well in ravines, i.e. area where the slope is much larger in a direction than the other. In such area, the SGD algorithm will oscillate in the direction of the large slope coefficient.

Momentum is a method that dampen these oscillations and push the solution in the right direction by adding a fraction γ of the update vector of the past time step to the current vector. It is defined with the following recursive formula:

$$v_n = \gamma v_{n-1} + \eta \nabla_\theta L_{\mathbf{x},\mathbf{y}}(\theta_{n-1})$$
$$\theta_n = \theta_{n-1} - v_n = \theta_{n-1} - \eta \nabla_\theta L_{\mathbf{x},\mathbf{y}}(\theta_{n-1}) - \gamma v_{n-1}$$

The damping parameter γ is usually set around 0.9.

In more practical words, the displacement v_n depends on the previous displacement v_{n-1} and on the slope $\nabla_\theta L_{\mathbf{x},\mathbf{y}}(\theta_{n-1})$: if they are of the same sign, the algorithm keeps going in the same direction. If they have different sign, the algorithm will slow down or even turn around: this mean indeed that the slope direction has changed, hence that the maximum is close.

Nesterov accelerated gradient

With the moment method, there is a risk that the maximum will be missed if the algorithm does not slows down soon enough.

The Nesterov accelerated gradient is an improvement on the momentum method: instead of computing the slope on the current point θ_{n-1}, the slope is computed slightly before, at $\theta_{n-1} - \gamma v_{n-1}$. This gives a sense of how the slope itself is evolving (hence the accelerated gradient denomination) and therefore allows a better anticipation of the .

$$v_n = \gamma v_{n-1} + \eta \nabla_\theta L_{\mathbf{x},\mathbf{y}}(\theta_{n-1} - \gamma v_{n-1})$$
$$\theta_n = \theta_{n-1} - v_n$$

The subsequent steps are the same as for the Momentum algorithm.

Adaptative methods

In the gradient descent algorithm, the learning rate η is constant. When trying to minimize $L_{\mathbf{x},\mathbf{y}}(\theta)$ however, it appears that the learning rate should depend on θ, i.e.
$\eta_n* = \text{argmin}_\eta(L_{\mathbf{x},\mathbf{y}}(\theta_n)) = \text{argmin}_\eta(\theta_{n-1} - \eta\nabla_\theta L_{\mathbf{x},\mathbf{y}}(\theta_{n-1}))$

The aim of adaptative methods is therefore to adapt the learning rate to every calculation step.

Adagrad

The Adagrad algorithm captures the step depency of the learning-rate by introducing G_n a vector the size of the parameter θ_n such that $G_n[i] = \sum_{t=0}^{n}\left(\frac{\partial L_{\mathbf{x},\mathbf{y}}}{\partial\theta_i}(\theta_t)\right)^2$.

At a given epoch n, the iteration step for the $i-th$ componet of the parameter vector is

$$\theta_{n,i} = \theta_{n-1,i} - \frac{\eta}{\sqrt{G_{n-1,i}+\epsilon}}\frac{\partial L_{\mathbf{x},\mathbf{y}}}{\partial\theta_i}(\theta_{n-1})$$

With $\epsilon > 0$ a smooting term so the iteration is always defined (usually $\epsilon = 10^{-8}$).

The underlying principle of this algorithm is to adapt the learning rate to the steepness of the learning slope. Indeed, the resulting learning rate $\frac{\eta}{\sqrt{G_{n-1,i}+\epsilon}}$ decrease when the slope coefficient $\frac{\partial L_{\mathbf{x},\mathbf{y}}}{\partial\theta_i}(\theta_t)$ increases in absolute terms.

Adadelta

The Adagrad algorithm tunes the learning rate in a meaningful way. However, as the denominator in the learning rate increases at every step (only positive terms are added), the overall learning rate may decrease to a point where no progress is made in the optimization.

Adadelta works based on the same principle of Adadelta, but prevents the learning rate from exceedingly decreasing by restricing the considered values to a fixed window of size w:$G_0[i] = \frac{1}{w}\sum_{t=0}^{w}\left(\frac{\partial L_{\mathbf{x},\mathbf{y}}}{\partial\theta_i}(\theta_t)\right)^2$.

Also, for improved efficiency, the w previous gradient are not stored the running average instead is calculated recursively:

$$G_n[i] = \gamma G_{n-1}[i] + (1-\gamma)\left(\frac{\partial L_{\mathbf{x},\mathbf{y}}}{\partial\theta_i}(\theta_n)\right)^2$$

with γ a pondering parameter of the current value and the past values (usually equal to 0.9).

The rest of the algorithm is then identical to Adagrad.

RMSprop

As for Adadelta, RMSprop aims at mitigating the learning rate decrease over the epochs in the Adagrad problem.

To that end, it also adapts the calculation of the vector G_n with an exponential decay of the gradient contributions over time (when Adadelta would cut off the values older than a given age) and taking the average of all contributions:

$G_n[i] = \frac{1}{n}\sum_{t=0}^{n}\left(\frac{\partial L_{\mathbf{x},\mathbf{y}}}{\partial \theta_i}(\theta_t)\right)^2 \exp(-\alpha(n-t))$.

with α a decaying parameter to be tuned by the user. The rest of the algorithm is identical to Adadelta.

ADAM

The Adaptative Moment Estimation (ADAM) is an alternative method to compute adaptative learning for each parameter.

In this method, both average of cumulative square-gradient G_n *and* and of cumulative gradient M_n are computed:

$$M_n[i] = \gamma_1 M_{n-1}[i] + (1-\gamma_1)\frac{\partial L_{\mathbf{x},\mathbf{y}}}{\partial \theta_i}(\theta_n)$$

$$G_n[i] = \gamma_2 G_{n-1}[i] + (1-\gamma_2)\left(\frac{\partial L_{\mathbf{x},\mathbf{y}}}{\partial \theta_i}(\theta_n)\right)^2$$

With both M_0 and G_0 initialized as vectors of zeroes, γ_1 and γ_2 close to 1 (usually 0.9 and 0.999 respectively).

As both the estimators M_n and G_n are biased, another estimator is calculated:

$$\hat{M_{n,i}} = \frac{M_{n,i}}{1-\gamma_1}$$

$$\hat{G_{n,i}} = \frac{G_{n,i}}{1-\gamma_2}$$

The iteration step is then adapted with the cumulative gradient M_n

$$\theta_{n,i} = \theta_{n-1,i} - \frac{\eta}{\sqrt{\hat{G_{n,i}} + \epsilon}}\hat{M}_{n,i}$$