A Fast Optimizer with Momentum and Sign

Peng-xu Jiang[1]

***Abstract*—We propose a fast optimization algorithm for deep learning that uses only the sign of momentum. This optimizer is as fast as Adam, but occupies half of the memory that Adam does. We also give an analysis of the effect of decay factor, which is the only hyper-parameter except for learning rate. We validate the analysis on the fashion-MNIST dataset.**

I. Background

A. Gradient Descent Method

For minimizing a smooth loss function $L: \mathbb{R}^n \to \mathbb{R}$, standard gradient descent method computes the gradient of L, and iterates along the negative direction of gradient so as to decrease L at each iteration. Explicitly, let $t \in \mathbb{N}$ denotes the step of iteration, thus the variable at step $t+1$ is given by (for each component α)

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

where the η is the learning rate. Since L is smooth, we have $L(\theta_{t+1}) < L(\theta_t)$ as long as the learning rate is sufficiently small and $\nabla L(\theta_t) \neq 0$. Thus, the iteration (1) always decrease L until reaching its (maybe local) minimum.

B. Momentum

Problems arise when applying gradient descent method directly to minimize the loss function computed on mini-batch. Because we feed mini-batch (rather than full-batch) to the model when we compute the loss function, denoted by L, there must be randomness in it. What we really want to minimize is the deterministic $\hat{L}$, the loss function computed on the full-batch. So, we hope that, iterated by the gradient descent method (1), the trajectory $(\theta_0, \theta_1, \dots)$ generated by ∇L (what we can compute) and the $(\hat{\theta}_0, \hat{\theta}_1, \dots)$ by $\nabla \hat{L}$ (what we expect to compute but cannot) share the same limit $\theta_\star$, the real best-fit value. Only when ∇L is sufficiently close to $\nabla \hat{L}$ can this be done, which indicates that we have to reduce the randomness from ∇L.

An efficient method for reducing randomness is averaging. Let $\{X_i | i = 1, \dots, n\}$ a set of i.i.d. random variables, each having variance $\mathrm{Var}[X]$. By central limit theorem, the variance of the averaged, $(1/n)\sum_i X_i$, is decreased by a factor $1/n$, thus $\mathrm{Var}[X]/n$. The same, we cache the most recent n gradients $\{\nabla L(\theta_{t-n+1}), \dots, \nabla L(\theta_t)\}$ at step t. Then, average over the cache to get the gradient used for iteration, $(1/n)\sum_{i=t}^{t-n+1} \nabla L(\theta_i)$. In this way, the variance of randomness in L is decreased by a factor $1/n$. By adjusting the value of n, the randomness can be limited sufficiently.

This "bare" average calls for caching the most recent gradients. It is very memory intensive when the dimension of θ goes high. A smarter one is moving average: given $\gamma \in [0, 1]$, the moving average of $\nabla L(\theta_t)$, denoted by g_t, is computed by iteration (for each component α)

$$g_t^{\alpha} = \gamma g_{t-1}^{\alpha} + (1 - \gamma)\, \nabla^{\alpha} L(\theta_t), \tag{2}$$

We initialize g_t by zero vector, thus $g_0^{\alpha} = 0$. The γ, called decay factor , determines how many old information of gradient, g_{t-1}, is to be "forgotten", and how many new information of gradient, $\nabla L(\theta_t)$, is to be "memorized". The g_t can be seen as a weighted average of $\{\nabla L(\theta_0), \dots, \nabla L(\theta_t)\}$, where the recent gradients have greater weights and the remote have less. Then, we iterate the θ_t by g_t instead of $\nabla L(\theta_t)$, as

$$\theta_{t+1}^{\alpha} = \theta_t^{\alpha} - \eta g_t^{\alpha}. \tag{3}$$

Moving average of gradient was first applied to gradient descent in 1986.[2] Later, the efficiency of moving average was usually explained as avoiding getting stucked by local minima. They compared moving average of gradient to the momentum in physics: the "heavy ball" rushes out of a local minimum with large "momentum". But, in a space with extremely high dimension, it is rare to encounter a local minimum, but saddle points instead.[3] So, this explanation cannot be faithful.

C. Sign of Gradient

The idea of using only the sign of gradient in optimization was first proposed by Martin Riedmiller and Heinrich Braun in 1992.[4] In their `rprop` (short for resilient back-propagation) algorithm, they only used the sign $|\nabla L(\theta_t)|$ (rather than the gradient $\nabla L(\theta_t)$ itself) in gradient descent iteration. This strategy helps release the issue where the components that are resistant to change (because their gradients have extremely small absolute values) slow down the training process. The `rprop` algorithm also adaptively tunes the learning rate to make it more stable, which, however, makes it impossible to deal with stochastic gradient.

Later in 2012, James Martens and Ilya Sutskever generalized the `rprop` algorithm to stochastic gradient and made it applicable to loss function with randomness (mini-batch). The new algorithm is called `rmsprop`.[5] As it is named, it employs root mean square (RMS for short) for approximating the sign of gradient, which helps stabilize the stochastic gradient. Explicitly, it iterates a variable s_t^{α} for each component θ with

$$s_t^{\alpha} = \beta s_{t-1}^{\alpha} + (1 - \beta)\, (\nabla^{\alpha} L(\theta))^2, \tag{4}$$

where $\beta \in (0, 1)$ and s is initialized by zero vector, thus $s_0^{\alpha} = 0$. It is another moving average with β the decay factor. Then, the iteration of gradient descent becomes (with ϵ a tiny number for

1. Email: shuiruge@whu.edu.cn

2. *Learning representations by back-propagating errors*, by David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams, 1986. DOI: 10.1038/323533a0.

3. *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization*, by Yann N. Dauphin and others, 2014. ArXiv: 1406.2572.

4. *Rprop - A Fast Adaptive Learning Algorithm*, by Martin Riedmiller und Heinrich Braun, 1992. DOI: 10.1109/ICNN.1993.298623.

5. Unpublished.

avoiding numerical error)

$$\theta_{t+1}^{\alpha} = \theta_t^{\alpha} - \eta \frac{\nabla^{\alpha} L(\theta_t)}{\sqrt{s_t^{\alpha} + \epsilon}}. \tag{5}$$

Finally, by combining `rmsprop` and momentum, the state-of-the-art optimizer, named `adam`, was proposed in 2014.[6] Explicitly, it first computes momentum by equation (2). Then, it computes the RMS by equation (4).

II. Method

A. The New Algorithm

We propose that, for computing θ_{t+1}, we shall use the sign of momentum, instead of using moving averaged RMS (5). That is,

$$\theta_{t+1}^{\alpha} = \theta_t^{\alpha} - \eta\, |g_t^{\alpha}|. \tag{6}$$

This is a combination of momentum, the strategy to reduce the randomness in the gradient caused by mini-batch, and of only using the sign, the strategy employed in `rprop` algorithm to force speeding up the optimization. By moving average, the g_t has been stabilized.

As a summary, we implement our method using `Numpy`.

```python
def mas(loss_gradient,
        initial_theta,
        iteration_steps,
        learning_rate=2e-4,
        decay_factor=0.95):
    """Optimizer␣with␣Momentum␣and␣Sign␣(MaS)."""
    # Initialization:
    theta = initial_theta
    g = np.zeros(np.shape(theta))
    # Iteration:
    for t in range(iteration_steps):
        # Moving average
        g = (
            decay_factor * g +
            (1-decay_factor) * loss_gradient(theta)
        )
        # Iterate by sign of gradient
        theta = theta - learning_rate * np.sign(g)
    return theta
```

B. Effect of Decay Factor

When γ is close to 0, the moving average g_t is easy to forget the old information of ∇L. Indeed, the factor $\gamma \ll 1$ in equation (2) implies $g_t \approx \nabla L(\theta_t)$, thus no averaging at all. So, for make the moving average effective, γ shall not be too small.

On the contrary, when γ is close to 1, the moving average g_t is hard to "accept" new information of ∇L_D. Indeed, the factor $(1-\gamma) \ll 1$ in equation (2) implies $g_t \approx g_{t-1}$ for all t. So, the moving average is hard to be modified when γ is close to 1. This is equivalent to a large learning rate, leading to an ascend of loss function instead of descent.

So, for efficiently and safely using moving average, the γ shall be moderate. And if, in practice, the randomness is so large that the moving average can be effective only when γ is close to 1, then we shall accordingly tune the learning rate η to be smaller, so as to decrease the loss function safely.

III. Experiments and Results

We make benchmark tests on fashion-MNIST dataset. We compare our method with the state-of-the-art optimizer, `adam`, as well as many other optimizers such as `adagrad`. The result can be found in figures 1 and 2.[7]

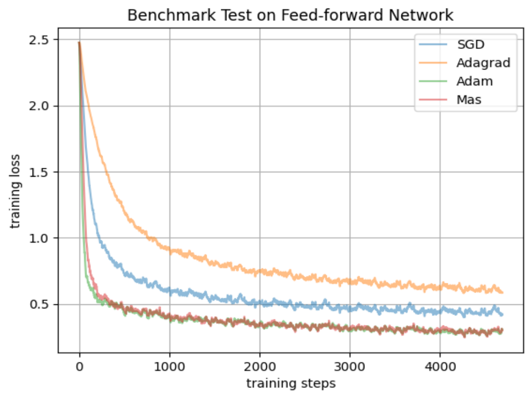

Figure 1. Comparing with other optimizers by training a feed-forward network on the training set of fashion-MNIST. Our method is denoted by "Mas". The feed-forward network contains two hidden layers, with 128 and 64 neurons respectively, and with ReLU activation. The output layer is linear. For hyper-parameters, we use the default values of each optimizer. The default learning rate of "Mas" is `2E-4`; and the default decay factor is `0.95`. For a better visualization, we smooth all the loss curves by moving average with decay factor 0.95. It can be seen that our method is as fast as `adam` optimizer, and out-performs the rest.

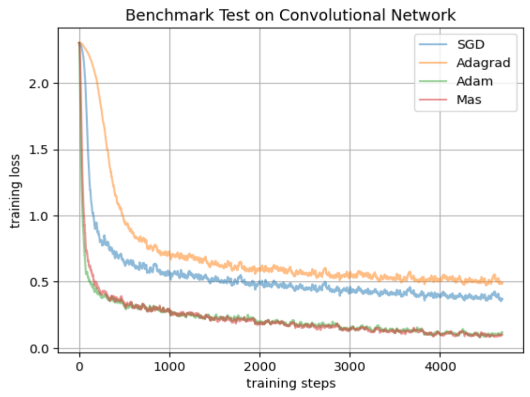

Figure 2. Comparing with other optimizers by training a convolutional network on the training set of fashion-MNIST. The convolutional contains layers in sequence: `Conv2D(32, 3)`, `ReLU()`, `Conv2D(64, 3)`, `ReLU()`, `MaxPool2D((2, 2))`, `Flatten()`, `Dense(128)`, `ReLU()`, and finally `Dense(10)` as the output layer. Again, for hyper-parameters, we use the default values of each optimizer. And again, it can be seen that our method is as fast as `adam` optimizer, and out-performs the rest.

6. *Adam: A Method for Stochastic Optimization*, by Diederik Kingma and Jimmy Ba, 2014. ArXiv: 1412.6980.

7. Code can be found in repository: https://github.com/shuiruge/mas-optimizer.

It is found that our method is as good as `adam` algorithm. But notice that we only cache g_t as variable, while adam additionally caches s_t, and that both g_t and s_t are vectors with the same dimension as θ, our method needs only half of the memory occupied by `adam` optimizer.

In the end, we also demonstrate the effect of decay factor γ in (2) in figure 3.

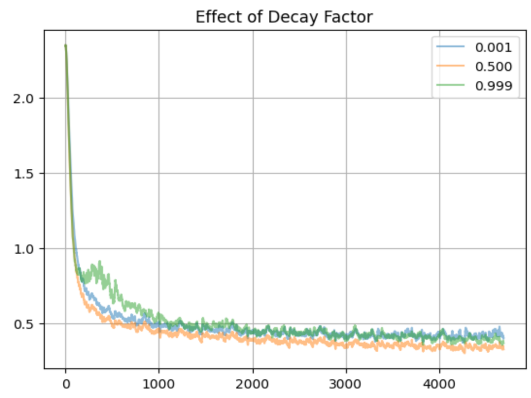

Figure 3. We demonstrate the effects of decay factors 0.001, 0.5, and 0.999. The optimization is made on a feed-forward network with the same architecture as in figure 1. We keep the learning rate default. It can be seen that a too small or too large decay rate (recall that decay rate is in $(0, 1)$) will slow down the optimization. Here, the best is the moderate (i.e. 0.5, the yellow line).

IV. Conclusion and Discussion

We have proposed a new algorithms that optimizes deep neural networks. From the basic benchmark tests, we have found that our method is as fast as the state-of-the-art `adam`, but occupies half of the memory that `adam` does. This is just the beginning. More benchmark tests are to be made on more complicated dataset and models.

V. Acknowledgements

We are grateful to Long Chen who for his very useful discussions, to Jun Guo who suggested to publish this method, and to Eden Li who supports me on every aspects.