ACCELERATING SGD WITH MOMENTUM FOR OVERPARAMETERIZED LEARNING REPRODUCE

Xiaocheng Zhong
Department of Electronics and Computer Science
Southampton University
Southampton, SO17 1BJ, UK
xz5m20@soton.ac.uk

Jinyu Tang
Department of Electronics and Computer Science
Southampton University
Southampton, SO17 1BJ, UK
jt3n20@soton.ac.uk

Tong Zhao
Department of Electronics and Computer Science
Southampton University
Southampton, SO17 1BJ, UK
tz1r20@soton.ac.uk

ABSTRACT

In Nesterov SGD, a Nesterov momentum is used to try to optimize the SGD method with momentum. But Nesterov momentum is sometimes not better than the ordinary SGD method of driving momentum, because it will have an effect that cannot be accelerated. In order to solve this problem, the Mass method introduces Nesterov SGD into a penalty term to achieve a truly faster effect than the standard momentum-driven SGD method. We have reproduced the gradient descent method of Mass, trying to prove that it has a better effect than SGD and SGD with momentum.

1 INTRODUCTION

We reproduce an optimizer using pytorch's method through the formula given in the paper *Accelerating SGD with momentum for over-parameterized learning* `https://openreview.net/forum?id=r1gixp4FPH`, test the optimizer using CNN and Fully connected neural network, and compare it with other optimizers.

Our work can be tracked on Github via
`https://github.com/COMP6248-Reproducability-Challenge/MaSS.git`. We ***completely*** reproduce the authors' work with Pytorch, while the code provided by the authors are implemented with Tensorflow and Keras.

2 OPTIMISER

First, we compare the difference between the standard SGD with momentum and the Nesterov SGD method, and then explore the changes made by MaSS to Nesterov SGD, and finally implement the optimizer.

2.1 COMPARISON OF STANDARD SGD WITH MOMENTUM AND NESTEROV SGD

Momentum simulates the inertia of object motion. SGD introduces momentum to make parameter updates retain the previously updated direction to a certain extent. At the same time, the current batch gradient is used to fine-tune the final direction, which can increase the descending speed. In order to further increase the descent speed, Nesterov SGD changes the momentum update method, updates one step according to the original update direction, then calculates the gradient at that position, and then uses this gradient value to correct the final update direction. The comparison of

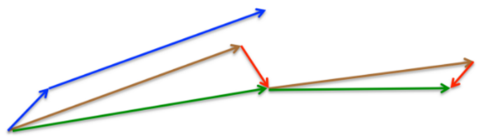


Figure 1: Momentum comparison

the two methods is shown in Figure 1. The blue line is the standard SGD with momentum update direction, the brown line is the momentum direction of Nesterov SGD, the red line is the correction momentum direction, and the green line is the actual update direction. Update comparison. It can be seen from this that the Nesterov SGD method makes the swing amplitude of the gradient update smaller, but this will reduce the magnitude of the momentum, making the Nesterov SGD no acceleration effect. Therefore Nesterov SGD is not a perfect improvement plan.

2.1.1 MaSS METHOD

MaSS (Momentum-added Stochastic Solver) is based on the idea of Nesterov SGD. A penalty term is added to the change of momentum to make Mass show exponential convergence during the decline. The emergence of this penalty term can avoid the problem of inability to accelerate caused by excessive correction in the process of momentum correction. A new penalty term coefficient is required before the correction term. Therefore, the penalty term can increase the descent speed while avoiding the problem of overcorrection of the momentum direction. The parameter update formula of Mass is shown in Figure2 by Liu & Belkin (2018), where u is the parameter that needs to be updated for gradient descent, **w** is a weight used to update u, and the blue part is the penalty term added first. The process of gradient update described by this formula does not reflect the change of momentum. In the paper, after transformation, Figure 3(a) is obtained. The right side of Figure 3(b) is the new parameter update formula, which is equivalent to the left side. **v** on the right side is momentum. During the updating process of **v**, the items underlined in red are newly introduced. Penalty items. In the code update process, use the logic on the right side of Figure 3(b) to update the parameters.

$$\begin{aligned} \mathbf{w}_{t+1} &\leftarrow \mathbf{u}_t - \eta_1 \tilde{\nabla} f(\mathbf{u}_t), \\ \mathbf{u}_{t+1} &\leftarrow (1+\gamma)\mathbf{w}_{t+1} - \gamma \mathbf{w}_t + \underline{\eta_2 \tilde{\nabla} f(\mathbf{u}_t)}. \end{aligned}$$

(a)

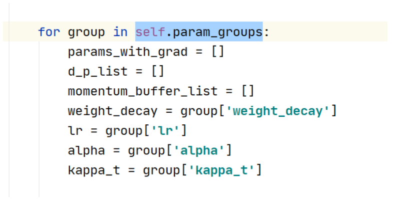
$$\begin{cases} \mathbf{w}_{t+1} \leftarrow \mathbf{u}_t - \eta_1 \tilde{\nabla} f(\mathbf{u}_t), \\ \mathbf{u}_{t+1} \leftarrow (1+\gamma)\mathbf{w}_{t+1} - \gamma \mathbf{w}_t + \eta_2 \tilde{\nabla} f(\mathbf{u}_t) \end{cases} \Longleftrightarrow \begin{cases} \mathbf{w}_{t+1} \leftarrow \mathbf{u}_t - \eta \tilde{\nabla} f(\mathbf{u}_t), \\ \mathbf{v}_{t+1} \leftarrow (1-\alpha)\mathbf{v}_t + \alpha \mathbf{u}_t - \underline{\delta \tilde{\nabla} f(\mathbf{u}_t)}, \\ \mathbf{u}_{t+1} \leftarrow \frac{\alpha}{1+\alpha}\mathbf{v}_{t+1} + \frac{1}{1+\alpha}\mathbf{w}_{t+1}. \end{cases}$$

(b)

Figure 2: (a) Initial formula (b) Transformation formula

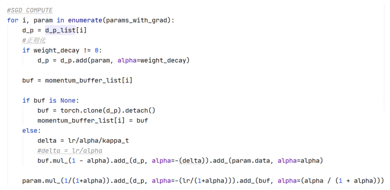
2.1.2 CODE IMPLEMENTATION PROCESS

We try to use Pytorch to implement an optimizer that uses the MaSS method to update parameters. First we define a class named MySGD, integrate the Optimizer in torch.optim. Rewrite the init, setstate and step methods. In the init method, I set the incoming parameters to the parameter list params that needs to be updated, the learning rate lr, and the weight attenuation. weight_decay, the alpha in the paper, and the parameter kappa_t to adjust the penalty item. First determine whether the parameter is less than zero. If it is less than zero, an exception will be thrown. In the setstate function, a state variable is passed in, and the state is updated using the setstate method of the parent class. The most important thing is to rewrite the step function, because it controls the way of gradient descent, first use a closure function to assign to loss. Read all the parameters in param_groups, param_groups is a dictionary in the Optimizer class, used to store all optimizers Parameters in the class. Respectively read all the parameters in param_groups, learning rate, alpha, kapaa_t, etc. The code is shown in Figure 3a. In the dictionary, there is an element named'params', which stores the parameters that need to be updated for gradient descent and the gradient of the parameter. The parameter value is added to the params_with_grad list for storage, and the gradient of the parameter is stored in Stored in the d_p_list list. The momentum of the previous stage needs to be stored in the momentum_buffer_list. Traverse params_with_grad to get the current parameters, gradient and speed. The penalty term coefficient added in the paper is lr/alpha/kata, so that all the required elements are obtained. Use the formula on the right side of Figure 2b to update the parameters, because **w** is this The parameters do not affect the gradient descent in the next stage, so I merged the formula of **w** into the update formula of the last u. The code for gradient calculation is shown in Figure 3b. Finally, the state of the optimizer is updated and loss is returned.



```
for group in self.param_groups:
    params_with_grad = []
    d_p_list = []
    momentum_buffer_list = []
    weight_decay = group['weight_decay']
    lr = group['lr']
    alpha = group['alpha']
    kappa_t = group['kappa_t']
```

(a)



```
#SGD COMPUTE
for i, param in enumerate(params_with_grad):
    d_p = d_p_list[i]
    #正则化
    if weight_decay != 0:
        d_p = d_p.add(param, alpha=weight_decay)

    buf = momentum_buffer_list[i]

    if buf is None:
        buf = torch.clone(d_p).detach()
        momentum_buffer_list[i] = buf
    else:
        delta = lr/alpha/kappa_t
        #delta = lr/alpha
        buf.mul_(1 - alpha).add_(d_p, alpha=-(delta)).add_(param.data, alpha=alpha)

    param.mul_(1/(1+alpha)).add_(d_p, alpha=-(lr/(1+alpha))).add_(buf, alpha=(alpha / (1 + alpha)))
```

(b)

Figure 3: (a) Get hyperparameters (b) Update method

3 Training with CNN Model

In this section, we ***completely*** reproduce the authors' work via Pytorch. We use convolutional neural networks (CNN) as models, combining MaSS, SGD, and SGD + Nesterov to train the Cifar-10 dataset for 60 epochs with learning rate $\eta = 0.1$, momentum $= 0.9$.

CIFAR-10 is a data set organized by Hinton students Alex Krizhevsky and Ilya Sutskever to identify universal objects. There are 10 categories of RGB color images: aircrafts, automobiles, birds, cats, deer, dogs, frogs, horses, boats and trucks. The pictures are 32×32 in size, with a total of 50,000 training tablets and 10,000 test pictures in the dataset.

According to the recommended structure of the paper, CNN has three convolutional layers with a kernel size of 5×5 and without padding, each layer convolution layer followed by a 2×2 max-pooling layer with the stride of 2. The first two layers have 64 channels, while the last layer has 128. After the last pooling layer, there is a fully connected ReLu-activated layer, with dropout probability at 0.5 and a softmax layer. The input data size of Cifar-10 is $3 \times 32 \times 32$. As calculated, the size is $64 \times 5 \times 5$ when the data passes the first two convolution layers and max-pooling layers. At this point, if passes the third layer convolution layer, the data size turns to $128 \times 1 \times 1$, which does not fit the requirements for max-pooling layer, so we abbandon the third max-pooling layer in practical applications (otherwise an error will appear).

When using this CNN model and SGD optimizer to train the dataset (with 20 epochs), we are surprised to find that the accuracy of the training set and test set are actually stabilized at around 0.1, and that indicates an underfitting, so the model needs to be improved. The training curves using the improved CNN model with the three optimizers are shown in Figure 4.

As Figure 4 shows, when MaSS is chosen as the optimizer, the train loss drops rapidly from 1.5 to about 0.2, but with the increase of epoch, it has a slow increase trend. This indicates that the model may be stuck in a local optimal solution. When SGD is selected as the optimizer, train loss can be quickly reduced from 1.7 to about 0.1, and then slowly decreases until it is about zero. This can be seen that the model performs well on the training set when SGD is used. And when we choose SGD-nesterov, train loss dropped from 1.5 to about 0.8, and then quickly increased to 2.2. The unusual performance of this case indicates that the model may have a degradation problem. Moreover, this divergence is also mentioned in the author's paper, which shows that some of the training will dissipate when the SGD-nesterov optimizer is used.

Although this experiment using CNN model training cifar10 did not fully reproduce the MaSS optimizer designed by the original author with the best performance in all aspects, it can still be seen in this diagram that Mass has the fastest loss drop rate, which is consistent with the original author's conclusion.

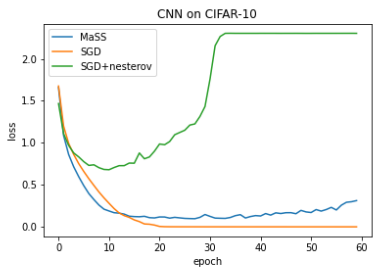


Figure 4: Training loss with CNN model

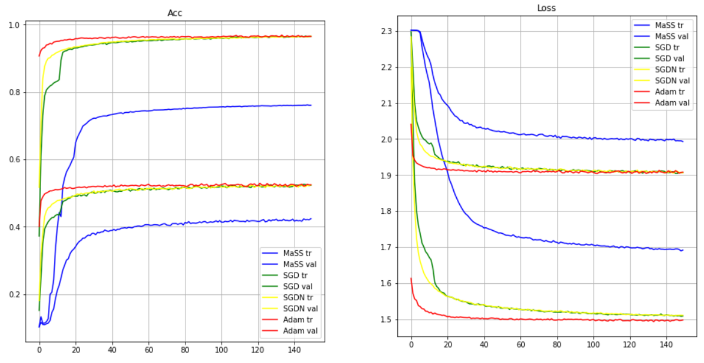
4 Training with fully connected model

In this section, we ***completely*** reproduce the authors' work via Pytorch and use fully connected networks as models, combining MaSS, SGD, SGD + Nesterov, and Adam to train the MNIST dataset for 150 epochs with learning rate $\eta = 0.01$, momentum $= 0.9$ and $\alpha = 0.05$, $\tilde{\kappa}_m = 3$ for MaSS as the paper recommended.

According to the paper, the fully-connected neural network has 3 hidden layers, with 100 ReLU-activated neurons in each layer. After each hidden layer, there is a dropout layer with keep probability of 0.5. The network takes 784-dimensional vectors as input and has 10 softmax-activated output neurons.

In Figure 5 it can be seen that all the optimizers get a convergent result after the 150 epochs. However, the MaSS performs worst among all the optimizers, with the worst convergence rate and worst final result. The proposed MaSS ends with a training accuracy at nearly 80% and validation accuracy at 42%, however, all the other optimizers end with training accuracy at about 95% and validation accuracy at about 52%. For training loss and validation loss, the MaSS ends with 1.7 and 2.0 respectively, while all the other three methods end with 1.5 and 1.9 respectively.

As shown previously, the MaSS does not perform better on MNIST, which is against the paper, but it also works to get a convergence result.



(a) (b)

Figure 5: Result of fully connected networks with different optimizers on MNIST

References

Chaoyue Liu and Mikhail Belkin. Accelerating sgd with momentum for over-parameterized learning. *arXiv preprint arXiv:1810.13395*, 2018.