Gradients Descent Optimization Algorithms

Yancheng Wang, Weihai Shen,

Team 39

***Abstract*—Gradient descent is increasingly popular for its success on machine learning algorithms based on neural network. Gradient descent optimization algorithms can be classified into three categories based on the volume of data used in each parameter update. In this article, we first compared these three different types of gradient descent optimization algorithms and addressed their pros and cons. Then we addressed four approaches that are broadly used to improve the performance of mini-batch gradient descent algorithm, namely, Polyak's classical momentum, Nesterov's Accelerated Gradient, RmsProp and ADAM. The four techniques are evaluated on their performance in a task of classification with a three-layer fully-connected neural network.**

***Index Terms*—Gradient descent, neural network, classification**

I. Introduction

ARTIFICIAL Neural Network, as a powerful machine learning method, has drawn great attention for its success on fields like computer vision, speech recognition and natural language processing. Recently, the performance of deep neural network based machine learning algorithms are comparable to and in some cases superior to human experts. Many neural network based machine learning algorithms can be regarded as the optimization of some scalar parameterized objective function requiring maximization or minimization with respect to its parameters. Gradient descent is the most popular optimization algorithm when the designed objective function is differentiable w.r.t its parameters.

Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or approximate gradient) of the function at the current point. Based on how much data we used to compute the gradient at each iteration, the gradient descent algorithms can be classified into three categories, namely, batch gradient descent, stochastic gradient descent (SGD) and mini-batch gradient descent [6]. Batch gradient descent computes the gradient of the cost function w.r.t to the parameters for the entire training dataset. While stochastic gradient descent performs a parameter update for each training example. Batch gradient descent is the safest way to do gradient descent since it can guarantee the convergence to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces. However, there could be similar samples in the whole dataset, batch gradient descent may do redundant works at each update. The huge volume of data also may do not fit in the memory we have for computation. Stochastic gradient descent can be much faster and can also be used to learn online. Actually, that's a trade-off between the accuracy of the parameter update and the time it takes to perform an update. Mini-batch gradient descent performs an update for every mini-batch of n training samples, which can be regarded as a compromise between batch gradient descent and stochastic gradient descent. The size of the mini-batch can vary for different applications. Mini-batch is usually our best choice for training a neural network. We also employ the term SGD for mini-batch gradient descent as it also does not use the whole dataset to perform parameter update each time.

Although mini-batch gradient descent seems to be more efficient and stable, it still does not guarantee a good convergence. There some more challenges for us to deal with. The choice of a proper learning rate can be really tricky. There is a trade-off between speed and accuracy. We also need to decide on how to adjust the learning rate for different phases of training and if the learning rate should be the same for all parameter. Additionally, when we need to optimize a highly non-convex error function, we need to avoid getting trapped in the numerous suboptimal local minima.

In this article, we will compare five different gradient descent optimization techniques to see if some techniques can help to deal with the aforementioned challenges. The comparisons we implemented are (i) No Momentum, (ii) Polyak's classical momentum [2], (iii) Nesterov's Accelerated Gradient [3], (iv) RmsProp [4] and (v) ADAM [5]. The implementation is based on the task of classifying the fashion-MNIST [1] dataset with a 3-layer fully-connected neural network.

II. Related Works

Gradient descent is the most popular optimization algorithm to optimize a neural network. Gradient descent is based on the observation that if the multi-variable function $F(x)$ is defined and differentiable in a neighborhood of a point a, then $F(x)$ decreases fastest if one goes from a in the direction of the negative gradient of F at a. It follows that, if

$$a_{n+1} = a_n - \eta \cdot \nabla F(a_n),$$

for $\eta \in \mathbb{R}_+$ small enough, then $F(a_n) \geq F(a_{n+1})$. With this observation, one starts with a guess point for a local minimum of F, and considers the sequence $a_0, a_1, a_2, \ldots$ such that

$$a_{n+1} = a_n - \eta \cdot \nabla F(a_n), n \geq 0.$$

Thus, we have a monotonic sequence

$$F(a_0) \geq F(a_1) \geq F(a_2) \geq \cdots,$$

and we hope this sequence converges to the desired local minimum.

When we apply gradient descent to optimize a machine

learning algorithm that based on neural network. The problem can always be demonstrated as optimizing an objective function $J(\theta)$ w.r.t its parameters $\theta \in R^d$. Then we update the model's parameters in the opposite direction of the gradient $\nabla_\theta F(\theta)$. The learning rate η determines the size of the steps we take.

When we apply gradient descent to optimize a model with real data. The first thing we need to think is how much data we use in each update. There is no doubt that if we use the whole data set every time., it is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces. However, if the size of the dataset used for training is really big. The optimization process may be very slow and the dataset may not fit in the computing resources we have. To deal with this issue, three popular gradient descent variants are broadly used for optimizing neural networks.

A. *Batch Gradient Descent*

Batch gradient descent [6] is the most intuitive way to update the model's parameter. In each update, we calculate the gradient for the whole dataset and update the parameters in the opposite direction of the gradient:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta).$$

However, batch gradient descent may do a lot of redundant computations for there might be many similar training samples as the training dataset can be really big. What's more it's also impossible if we want to add additional training samples in an online manner during the training process.

B. *Stochastic Gradient Descent*

Stochastic gradient descent (often shortened to SGD) [6], also known as incremental gradient descent, is an iterative method for optimizing a differentiable objective function, a stochastic approximation of gradient descent optimization. SGD updates the model's parameters for each training example $x^{(i)}$ and $y^{(i)}$:

$$\theta = \theta - \eta \cdot \nabla_\theta J\left(\theta; x^{(i)}, y^{(i)}\right).$$

SGD avoids the redundant computation problems as it performs update for parameters with one training sample a time. Consequently, it can be much faster to update with SGD and can be used to perform online training with added training samples during the training process.

C. *Mini-batch Gradient Descent*

Comparing batch gradient descent and SGD, we need to make a trade-off between the accuracy of the parameter update and time cost to update the parameter. Mini-batch gradient descent [6] is a compromise between batch gradient descent and SGD. Mini-batch gradient takes n training samples from the whole training batch each time for updating the parameter of the model:

$$\theta = \theta - \eta \cdot \nabla_\theta J\left(\theta; x^{(i:i+n)}, y^{(i:i+n)}\right).$$

With more training samples involved in each update, mini-batch gradient descent can achieve more stable convergence. The efficiency of the update can also be much better than batch gradient descent. Common mini-batch sizes range between 50 and 256, but can vary for different applications. Mini-batch gradient descent is basically our best choice when training a neural network. We also use the term SGD for mini-batch gradient descent.

III. Methods

As deep learning being widely used, we can always encounter machine learning tasks dealing with a huge amount of training data. The parameter volume of the neural network we use is also much bigger. SGD greatly improved the parameter update efficiency to solve deep learning tasks. However, it also brings us some new problems. It's really hard to choose a proper learning rate. Setting this parameter too high can cause the algorithm to diverge. While setting it too low makes it slow to converge. A conceptually simple extension of stochastic gradient descent makes the learning rate a decreasing function of the iteration number t, giving a learning rate schedule, so that the first iteration causes large change in the parameters, while the later ones do only fine-tuning.

A. *Polyak's classical Momentum*

Momentum [2] is a method that helps accelerate SGD in the relevant direction and dampens. It's a really intuitive way to deal with the aforementioned problem. It does this by adding a fraction of the update vector of the past time step to the current update vector:

$$v_t = \gamma v_{t-1} + \eta \cdot \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$

The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions.

B. *Nesterov Accelerated Momentum*

Nesterov accelerated gradient (NAG) [3] is a way to give our momentum a notion of where it is going so that it knows to slow down before the hill slopes up again. In this way we're first looking at a point where current momentum is pointing to and computing gradients from that point:

$$v_t = \gamma v_{t-1} + \eta \cdot \nabla_\theta J(\theta - \gamma v_{t-1})$$

$$\theta = \theta - v_t$$

It is much clearer when we look at the following picture.

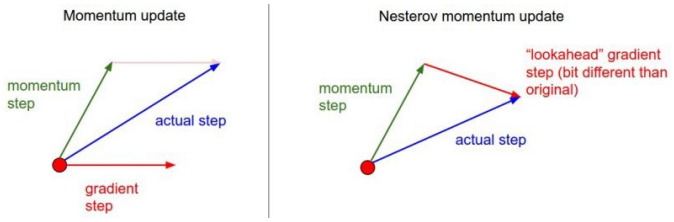
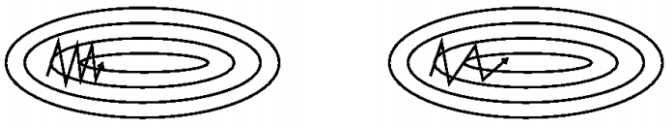


Fig. 1. Nesterov update (Source: G. Hinton's lecture 6c)

With Stochastic Gradient Descent we do not need to compute the exact derivate of our loss function. Instead, we're estimating it on a small batch. Which means we are not always

going in the optimal direction, because our derivatives are 'noisy'. So, exponentially weighed averages can provide us a better estimate which is closer to the actual derivate than our noisy calculations. This is one reason why momentum might work better than classic SGD.



(a) SGD without momentum (b) SGD with momentum

Fig. 2. SGD with& without momentum (Source: Genevieve B. Orr)

The other reason lies in ravines. Ravine is an area, where the surface curves much more steeply in one dimension than in another. Ravines are common near local minimum in deep learning and SGD has troubles navigating them. SGD will tend to oscillate across the narrow ravine since the negative gradient will point down one of the steep sides rather than along the ravine towards the optimum. Momentum helps accelerate gradients in the right direction. This is expressed in Fig.2.

C. *RMSProp*

RMSProp (Root Mean Square Propagation) [4] is a method in which the learning rate is adapted for each of the parameters. The idea is to divide the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight. So, first the running average is calculated in terms of means square,

$$v_t = \gamma v_{t-1} + (1-\gamma)\cdot(\nabla_\theta J(\theta))^2$$

where, γ is the forgetting factor.
And the parameters are updated as,

$$\theta = \theta - \frac{\eta}{\sqrt{v_t + \epsilon}}\nabla_\theta J(\theta)$$

Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

D. *Adam*

Adam [5] is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v like RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\cdot\nabla_\theta J(\theta)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\cdot(\nabla_\theta J(\theta))^2$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively. However, m_t and v_t are found to be biased towards zero during the training process, especially when β_1 and β_2 are close to 1. To deal with this problem, a bias-corrected version of m_t and v_t is computed:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_1^t}$$

Then m_t and v_t are used to update the parameters:

$$\theta = \theta - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}}\hat{m}_t$$

The authors of Adam propose default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ.

IV. Experiments

Dataset. In our experiments, we used the Fashion-MNIST [1] dataset, which is created to serve as a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits. It consists of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. Some examples of the Fashion-MNIST dataset are showed in Fig.3.

Fig. 3. Fashion-MNIST dataset

Setup. In our experiments, the whole training set is used for training and the whole test set is used for test. We implemented a three-layer neural network for classification. The number of nodes in each hidden layer is 500, 100, and 10. The structure of the neural network we implemented is showed in Fig.4. Mini-batch gradient descent approach is used to optimize the neural network. Two batch size 100 and 1000 are tested separately. Five parameters updating policy (i)No Momentum, (ii)Polyak's classical momentum, (iii)Nesterov's Accelerated Gradient, (iv)RmsProp and (v)ADAM are implemented respectively. The learning rate for No Momentum, Polyak's classical momentum and Nesterov's Accelerated Gradient is set to 0.1. The parameter γ for Polyak's classical momentum and Nesterov's Accelerated Gradient is set to 0.9. The learning rate for RmsProp and ADAM is set to 0.01. The parameter γ and ϵ for RmsProp is set to 0.9 and 10^{-8} respectively. The parameter β_1, β_2 and ϵ for Adam is set to 0.9, 0.999 and 10^{-8} respectively.

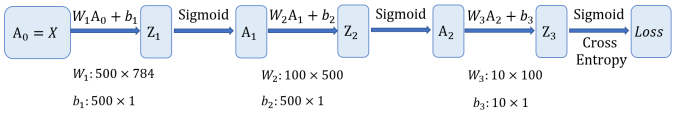


Fig. 4. Neural network structure for the experiments

Results. The loss during the training process for our experiments is shown in Fig. 5 and Fig.6. As we can see from the figures, with the help of optimization approaches, the training loss decreases much faster during the training process, especially at the beginning of it. However, as we optimize our models in SGD manner, the variation of the training loss can be a little higher with a smaller batch size. It can be much better when we increase the batch size for each update.

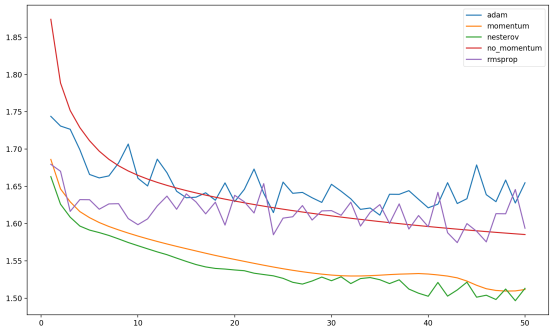


Fig. 5. Training loss (batch size = 100)

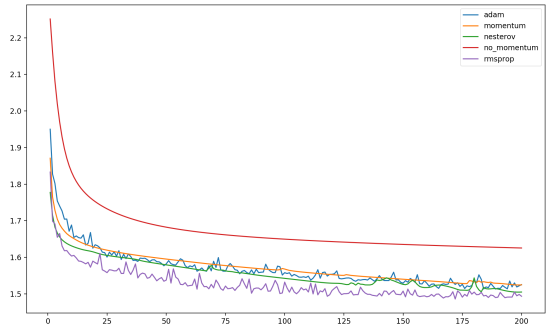


Fig. 6. Training loss (batch size = 1000)

The accuracy of the trained models with different setups is displayed in table 1. With the help of our optimization approaches, the final test accuracy is increased. Polyak's classical momentum achieve the highest accuracy for both the batch size of 100 and 1000.

TABLE I
TEST ACCURACY WITH DIFFERENT SETUP

Batch Size	No Momentum	Polyak's classical momentum	Nesterov's Accelerated Gradient	RmsProp	ADAM
100	0.8778	**0.8836**	0.8823	0.8638	0.8758
1000	0.8623	**0.8914**	0.8903	0.8854	0.8879

V. CONCLUSIONS

In this article, we first addressed three different types of gradient descent algorithms. As mini-batch gradient descent being the most popular algorithm to optimize neural network, we illustrated four optimization techniques for mini-batch gradient descent (also referred as SGD). The experimental results on the task of classification in fashion-MNIST dataset shows that the illustrated dataset can improve the optimization performance of SGD in convergence speed and final test accuracy.

REFERENCES

[1] H. Xiao, K. Rasul, and R. Vollgraf. (2017) Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.
[2] B. T. Polyak, “Some methods of speeding up the convergence of iteration methods,” USSR Computational Mathematics and Mathematical Physics, vol. 4, no. 5, pp. 1–17, 1964.
[3] Y. E. Nesterov, “A method for solving the convex programming problem with convergence rate o (1/kˆ 2),” in Dokl. Akad. Nauk SSSR, vol. 269, 1983, pp. 543–547.
[4] T. Tieleman and G. Hinton, “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude,” COURSERA: Neural networks for machine learning, vol. 4, no. 2, pp. 26–31, 2012.
[5] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” arXiv preprint arXiv:1412.6980, 2014.
[6] Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747, 2016.

DIVISION-WORK

Implementing the basic 3-layer fully connected network without momentum -**Weihai Shen**
Implementing gradient descent optimization techniques Polyak's classical momentum and Nesterov's Accelerated Gradient -**Weihai Shen**
Implementing gradient descent optimization techniques RmsProp and ADAM -**Yancheng Wang**
Writing the article -**Yancheng Wang**
Preparing for the final presentation -**Weihai Shen**

SELF-PEER EVALUATION

Yancheng Wang - 20	My self - 20