华南理工大学

South China University of Technology

The Experiment Report of Machine Learning

School: School of Software Engineering

Subject: Software Engineering

Author:

Xiaoli Tang

Supervisor:

Qingyao Wu

Student ID:

201720145044

Grade:

Graduate

December 14, 2017

Logistic Regression, Linear Classification and Stochastic Gradient Descent

Abstract — **Gradient descent optimization algorithms, while increasingly popular, are often used as black-box optimizers, as practical explanations of their strengths and weaknesses are hard to come by. This experiment aims to provide the reader with intuitions with regard to the behavior of different algorithms that will allow her to put them to use. In this experiment, we look at different variants of gradient descent, summarize challenges, introduce the most common optimization algorithms.**

i. INTRODUCTION

Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks. At the same time, every state-of-the-art Deep Learning library contains implementations of various algorithms to optimize gradient descent. These algorithms, however, are often used as black-box optimizers, as practical explanations of their strengths and weaknesses are hard to come by.

This experiment post aims at providing us with intuitions towards the behavior of different algorithms for optimizing gradient descent that will help us put them to use. We are first going to introduce the most common optimization algorithms by showing their motivation to resolve these challenges and how this leads to the derivation of their update rules. And then, we will set up different experiments to compare the most common optimization algorithms. Finally, we will give a conclusion to summarize this experiment.

ii. METHODS AND THEORY

The loss function of Logistic Regression and Linear Classification are as shown as formula 1 and 2 separately:

$$L = -\frac{1}{N}\sum_{i=1}^{N}[y_i log\frac{1}{1+e^{-W^T x_i}} + (1 - y_i)log\frac{1}{1+e^{-W^T x_i}}] \quad (1)$$

$$L = \frac{1}{N}\sum_{i=1}^{N}[\frac{\lambda}{2}||W||^2 + \max(0, 1 - y_i W^T x_i)] \quad (2)$$

The gradient of the weight are as follows:

$$g = \frac{1}{m}X^T(\frac{1}{1+e^{-W^T x_i}} - y) \quad (3)$$

$$g = \begin{cases} \frac{1}{N}\sum_{i=1}^{N}\lambda W, & y_i W^T x_i > 1 \\ \frac{1}{N}\sum_{i=1}^{N}[\lambda W - x_i^T y_i], & y_i W^T x_i \leq 1 \end{cases} \quad (4)$$

A. Stachastic Gradient Descent

Stachastic gradient descent (SGD) in contrast performs a parameter update for each training example x^i and label y^i:

$$\theta = \theta - \nabla_\theta J(\theta; x^i; y^i) \quad (5)$$

Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online.

SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily as in Fig. 1.

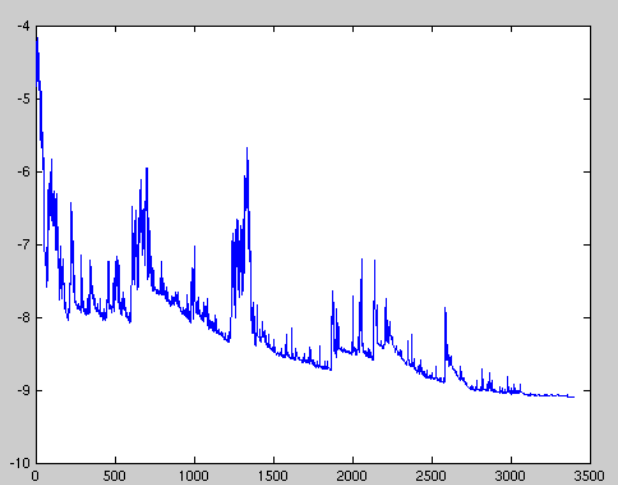

Fig.1. SGD fluctuation

While batch gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima. On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting. However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behavior as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively. Its code fragment simply adds a loop over the training examples and evaluates the gradient w.r.t. each example.

B. Momentum

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another[1], which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress. along the bottom towards the local optimum as in Fig. 2.

Fig.2. SGD without momentum

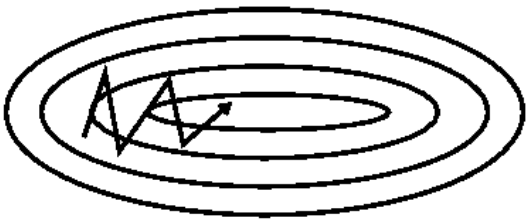

Fig.3. SGD with momentum

Momentum[2] is a method that helps accelerate SGD in the relevant direction and dampens oscillations as can be seen in Fig. 3. It does this by adding a fraction γ of the update vector of the past time step to the current update vector:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \quad (6)$$

$$\theta = \theta - v_t \quad (7)$$

the momentum term γ is usually set to 0.9 or a similar value.

Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. $\gamma < 1$). The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions.

As a result, we gain faster convergence and reduced oscillation.

C. Nesterov Accelerated Gradient

However, a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory. We'd like to have a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.

Nesterov accelerated gradient (NAG) [3] is a way to give our momentum term this kind of prescience. We know that we will use our momentum term γv_{t-1} to move the parameters θ. Computing $\theta - \gamma v_{t-1}$ thus gives us an approximation of the next position of the parameters (the gradient is missing for the full update), a rough idea where our parameters are going to be. We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \quad (8)$$

$$\theta = \theta - v_t \quad (9)$$

Again, we set the momentum term γ to a value of around 0.9. While Momentum first computes the current gradient (small blue vector in Fig. 4) and then takes a big jump in the direction of the updated accumulated gradient (big blue vector), NAG first makes a big jump in the direction of the previous accumulated gradient (brown vector), measures the gradient and then makes a correction (red vector), which results in the complete NAG update (green vector). This anticipatory update prevents us from going too fast and results in increased responsiveness, which has significantly increased the performance of RNNs on a number of tasks [4].

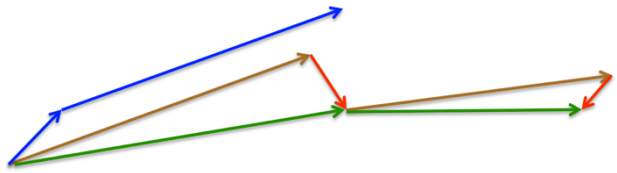

Fig.4. Nesterov update

D. Adadelta

Adadelta [5] is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size w.

Instead of inefficiently storing w previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average $E[g^2]_t$ at time step t then depends (as a fraction γ similarly to the Momentum term) only on the previous average and the current gradient:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \quad (10)$$

We set γ to a similar value as the momentum term, around 0.9. For clarity, we now rewrite our vanilla SGD update in terms of the parameter update vector $\Delta\theta_t$:

$$\Delta\theta_t = -\eta \cdot g_{t,i} \quad (11)$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t \quad (12)$$

The parameter update vector of Adagrad that we derived previously thus takes the form:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t+\varepsilon}} \odot g_t \quad (13)$$

We now simply replace the diagonal matrix G_t with the decaying average over past squared gradients $E[g^2]_t$:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t+\varepsilon}} g_t \quad (14)$$

As the denominator is just the root mean squared (RMS) error criterion of the gradient, we can replace it with the criterion short-hand:

$$\Delta\theta_t = -\frac{\eta}{RMS[G]_t} g_t \quad (15)$$

The authors note that the units in this update (as well as in SGD, Momentum, or Adagrad) do not match, i.e. the update should have the same hypothetical units as the parameter. To realize this, they first define another exponentially decaying average, this time not of squared gradients but of squared parameter updates:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2 \quad (16)$$

The root mean squared error of parameter updates is thus:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \varepsilon} \quad (17)$$

Since $RMS[\Delta\theta]_t$ is unknown, we approximate it with the RMS of parameter updates until the previous time step. Replacing the learning rate η in the previous update rule with $RMS[\Delta\theta]_{t-1}$ finally yields the Adadelta update rule:

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t \quad (18)$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t \quad (19)$$

With Adadelta, we do not even need to set a default learning rate, as it has been eliminated from the update rule.

E. RMSprop

RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his Coursera Class.

RMSprop and Adadelta have both been developed independently around the same time stemming from the need to resolve Adagrad's radically diminishing learning rates. RMSprop in fact is identical to the first update vector of Adadelta that we derived above:

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2 \quad (20)$$

$$\theta_{t+1} = \theta_t - -\frac{\eta}{\sqrt{E[g^2]_t+\varepsilon}} g_t \quad (21)$$

RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients. Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

F. Adam

Adaptive Moment Estimation (Adam) [6] is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \quad (22)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \quad (23)$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).

They counteract these biases by computing bias-corrected first and second moment estimates:

$$\widehat{m}_t = \frac{m_t}{1-\beta_1^t} \quad (24)$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \quad (25)$$

They then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - -\frac{\eta}{\sqrt{\hat{v}_t}+\varepsilon}\widehat{m}_t \quad (26)$$

The authors propose default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ε. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

iii. EXPERIMENTS

A. Dataset

Experiment uses a9a of LIBSVM Data, including 32561/16281(testing) samples and each sample has 123/123 (testing) features.

B. Implementation

a) Logistic Regression and Sochastic Gradient Descent

1. Model and loss function

The sigmoid function is used in this experiment as score function while logarithmic loss function is chosen to be the loss function.

2. Hyper-parameters

We fixed some hyper-parameters and used exhaustive grid search to tuning the hyper parameter to find the best estimator with different optimize algorithm. The fixed hyper-parameters are epoch and batch_size:

- Epoch = 5, means that we used the whole training data set 5 times to perform the gradient decent.
- Batch_size = 128, means that we divided the training data into 128 batches.
- The hyper-parameters' candidate are as follows:

	SGD	**NAG**	**RMSProp**	**AdaDelta**	**Adam**
η	0.01	0.01	0.01	0.01	0.01
	0.1	0.1	0.1	0.1	0.1
	0.2	0.2	0.2	0.2	0.2
	0.3	0.3	0.3	0.3	0.3
	0.4	0.4	0.4	0.4	0.4
γ		0.5	0.5	0.5	

		0.7 0.9 0.99	0.7 0.9 0.99	0.7 0.9 0.99	
β_1					0.5 0.7 0.9 0.99 0.999
β_2					0.5 0.7 0.9 0.99 0.999

The performance of best estimators with five different optimize algorithm are as follows:

	SGD	**NAG**	**RMSProp**	**AdaDelta**	**Adam**
η	0.2	0.2	0.01	0.01	0.01
γ		0.9	0.9	0.9	
β_1					0.999
β_2					0.999
Acc	0.758	0.763	0.764	0.764	0.764
uracy					
Loss	0.227	0.231	0.226	0.220	0.218

3. Results of the algorithms on logistic regression

The loss of train data set and validation data set with SGD, NAG, RMSProp，AdaDelta and Adam are as follows:

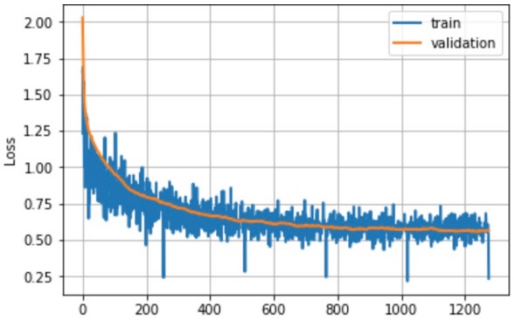

Fig.5. Loss of SGD

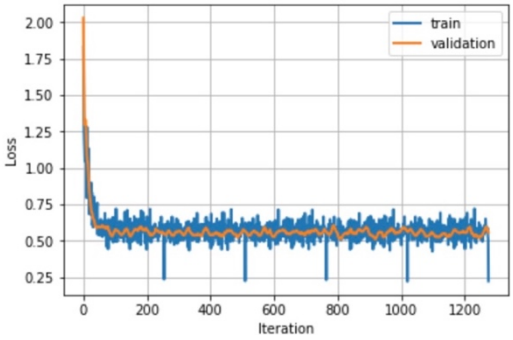

Fig.6. Loss of NAG

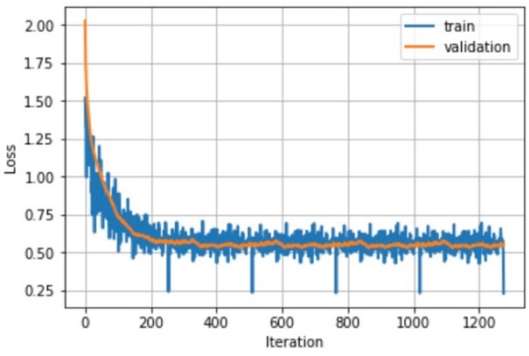

Fig.7. Loss of RMSProp

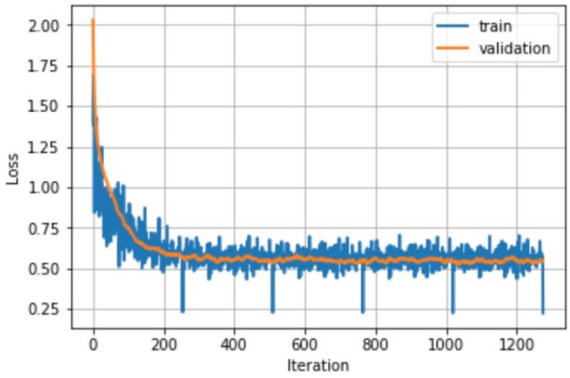

Fig.8. Loss of AdaDelta

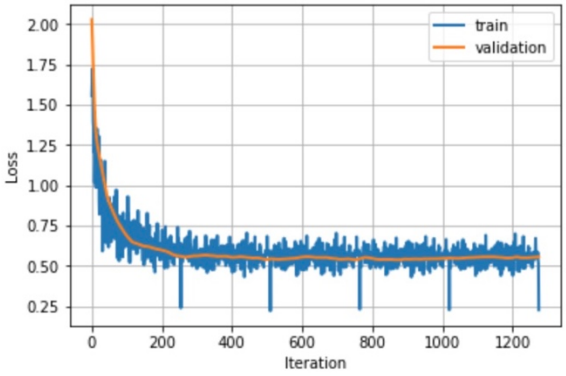

Fig.9. Loss of Adam

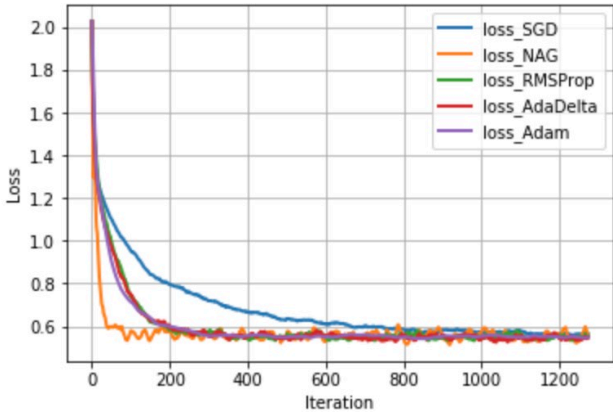

Fig.10. Loss Comparison

Fig.5, Fig.6, Fig.7, Fig.8, Fig.9, Fig.10 show that the loss of validation data set tends to be stable and tend to be similar results after a certain number of iterations. But the loss of training data set is oscillating, generally stable and dropping. Among all the algorithms, NAG's loss of validation data set drops fastest, while SGD's drops most slowly.

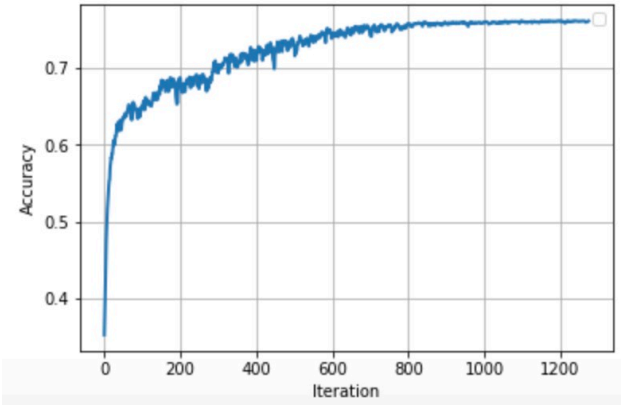

Fig.11. Accuracy of SGD

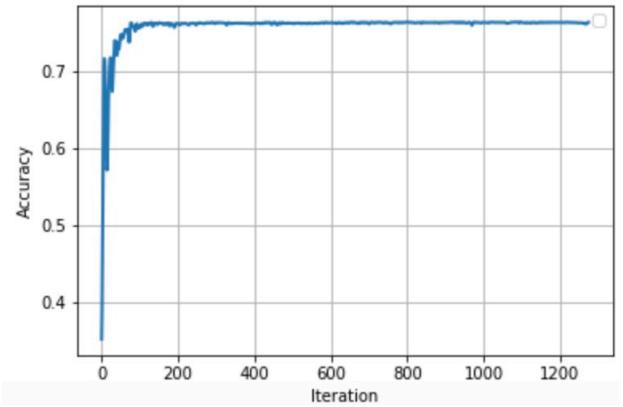

Fig.12. Accuracy of NAG

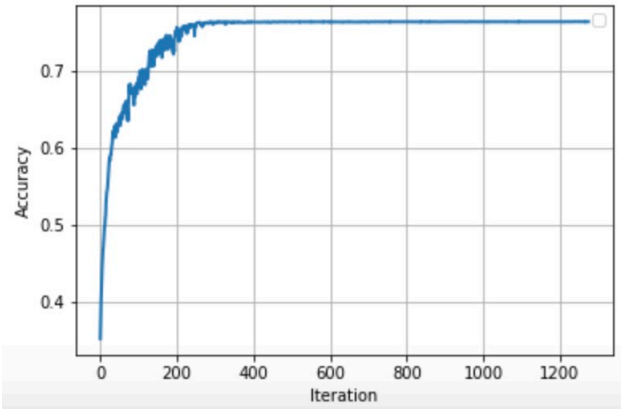

Fig.13. Accuracy of RMSProp

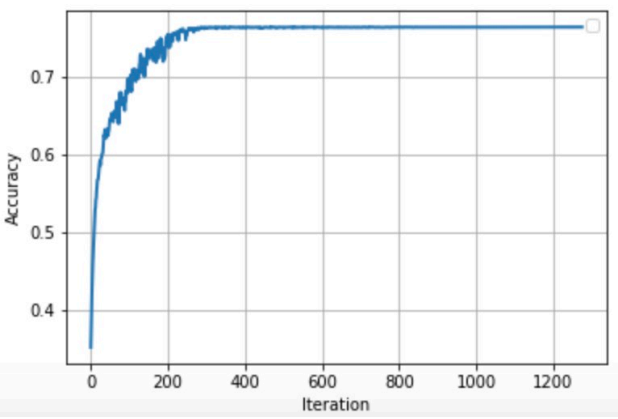

Fig.14. Accuracy of AdaDelta

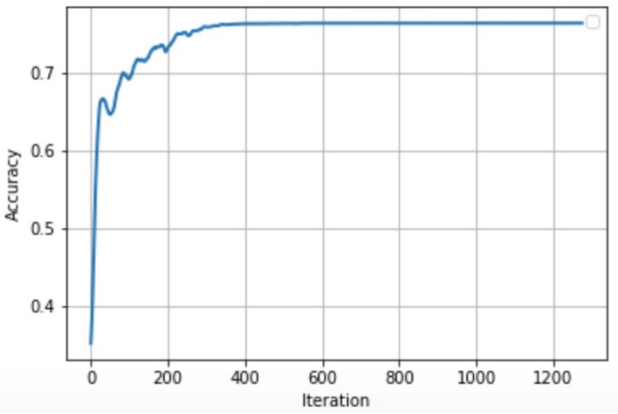

Fig.15. Accuracy of Adam

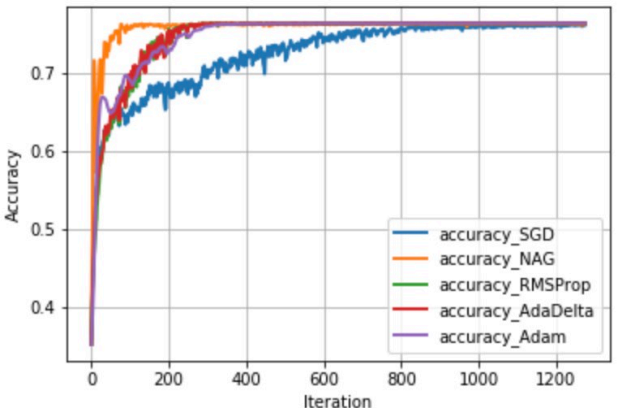

Fig.16. Accuracy Comparison

We can see that the accuracy of validation data set tends to be stable and tend to be similar results after a certain number of iterations. Among all the algorithms, NAG's accuracy of validation data set increases at the highest speed, while SGD's increases most slowly.

So, in my logistic regression, NAG is the best algorithm considering the loss and the accuracy.

b) Linear Classification and Stochastic Gradient Descent

1. Model and loss function

The SVM is used to implement the linear claasifier while hinge loss is chosen to be the loss function.

2. Hyper-parameters

We fixed some hyper-parameters and used exhaustive grid search to tuning the hyper parameter to find the best estimator with different optimize algorithm. The fixed hyper-parameters are epoch and batch_size:

- Epoch = 2, means that we used the whole training data set 2 times to perform the gradient decent.
- Batch_size = 128, means that we divided the training data into 128 batches.
- The hyper-parameters' candidate are as follows:

	SGD	**NAG**	**RMSProp**	**AdaDelta**	**Adam**
η	0.01	0.01	0.01	0.01	0.01
	0.1	0.1	0.1	0.1	0.1
	0.2	0.2	0.2	0.2	0.2

	0.3 0.4	0.3 0.4	0.3 0.4	0.3 0.4	0.3 0.4
λ	0.01 0.1 0.2 0.3 0.4	0.01 0.1 0.2 0.3 0.4	0.01 0.1 0.2 0.3 0.4	0.01 0.1 0.2 0.3 0.4	0.01 0.1 0.2 0.3 0.4
γ		0.5 0.7 0.9 0.99	0.5 0.7 0.9 0.99	0.5 0.7 0.9 0.99	
β_1					0.5 0.7 0.9 0.99 0.999
β_2					0.5 0.7 0.9 0.99 0.999

The performance of best estimators with five different optimize algorithm are as follows:

	SGD	**NAG**	**RMSProp**	**AdaDelta**	**Adam**
η	0.2	0.2	0.01	0.01	0.01
λ	0.1	0.1	0.1	0.1	0.1
γ		0.9	0.9	0.9	
β_1					0.999
β_2					0.999
Accuracy	0.764	0.763	0.763	0.764	0.763
Loss	2.487	2.435	2.238	2.344	2.411

3. Results of the algorithms on linear classification

Fig. 17-21 show the result of SGD, NAG, RMSProp, AdaDelta, Adam, which include the loss of the training set and validation set. Fig. 23-27 show the accuracy of the validation set.

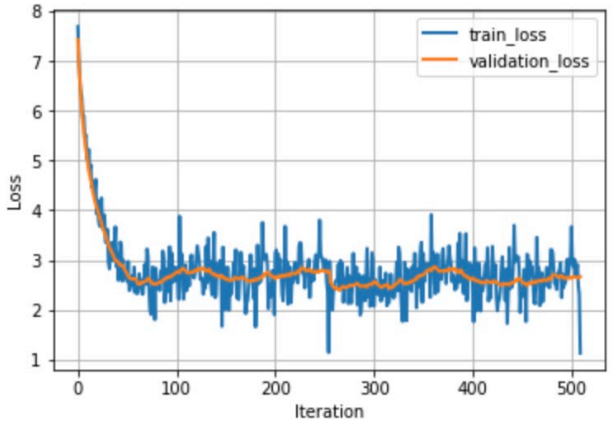

Fig.17. Loss of SGD

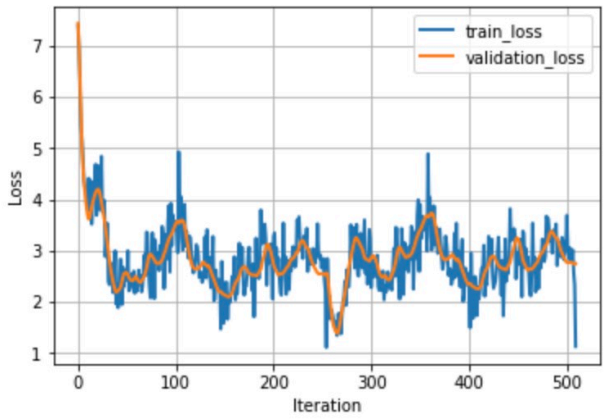

Fig.18. Loss of NAG

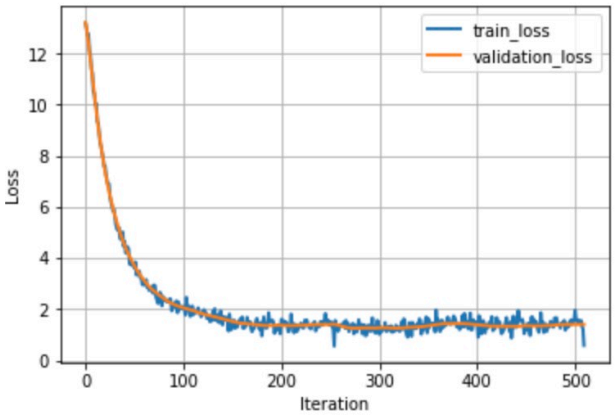

Fig.21.Loss of Adam

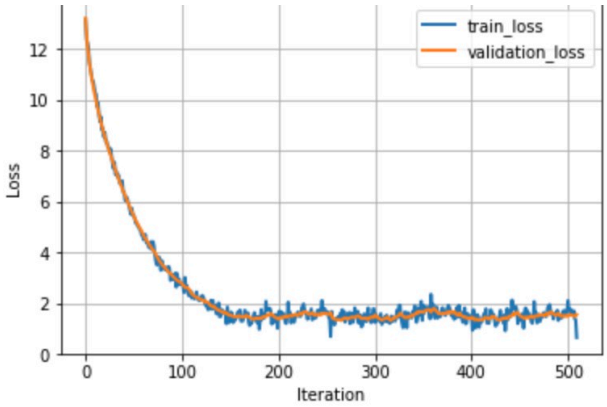

Fig.19. Loss of RMSProp

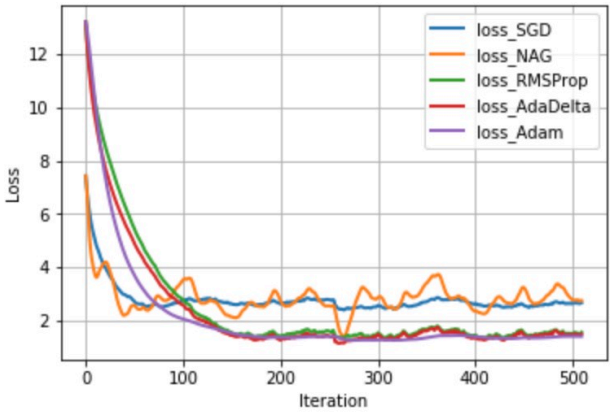

Fig.22. Loss Comparison

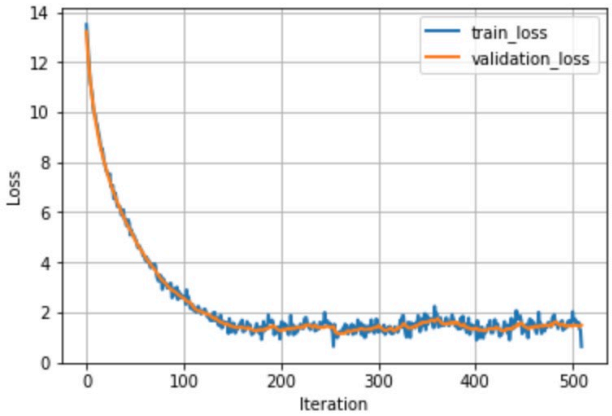

Fig.20. Loss of AdaDelta

Above Figs show that the loss of validation data set tends to be dropping after a certain number of iterations generally. NAG's fluctuates the most. And the loss of training data set is oscillating, generally stable and dropping. Among all the algorithms, Adam's loss of validation data set drops sharply, while NAG's drops most slowly.

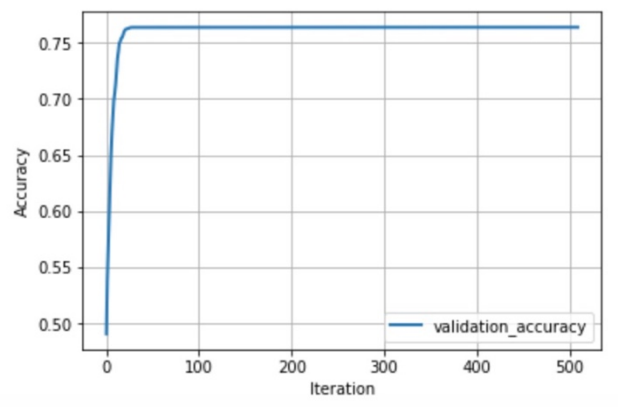

Fig.23. Accuracy of SGD

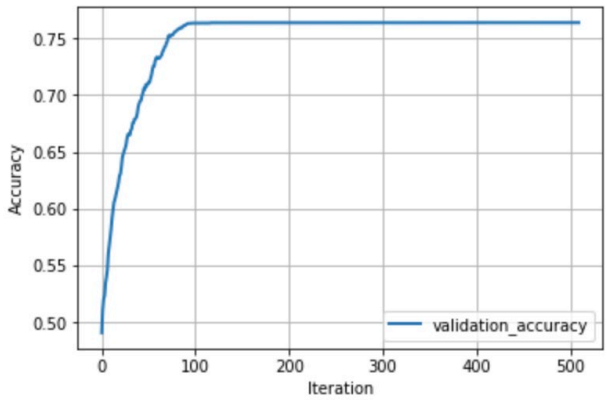

Fig.26. Accuracy of AdaDelta

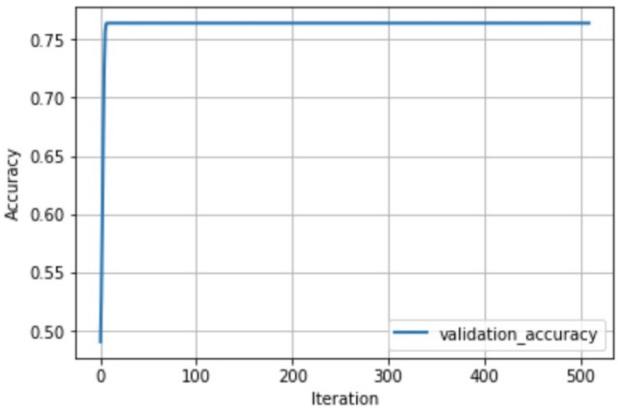

Fig.24. Accuracy of NAG

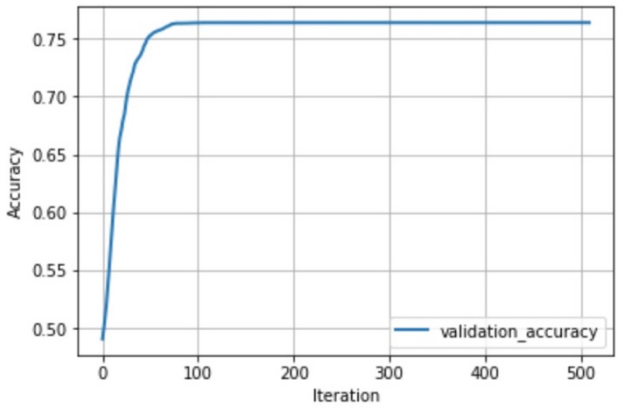

Fig.27. Accuracy of Adam

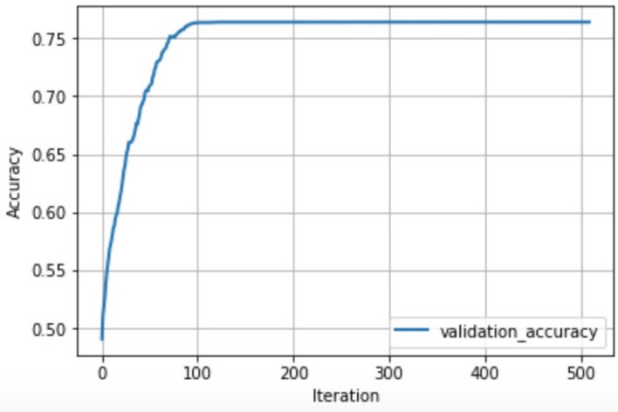

Fig.25. Accuracy of RMSProp

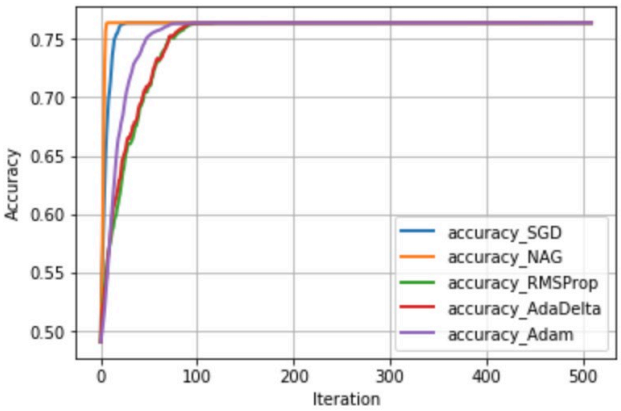

Fig.28. Accuracy Comparison

It's obvious that the accuracy of validation data set tends to be stable and tend to be similar results after a certain number of iterations. Among all the algorithms, NAG's accuracy of

validation data set increases at the highest speed, while RMSProp's increases most slowly.

C. Analysis

The result of the Logistic Regression shows that NAG is usually good at find a minimum, and it converges sharply. And for Linear Classification, Adam usually achieves to find a minimum, but if might take a long time to converge. SGD is good at converging sharply, but ends up with a more disappointing result compared with Adam.

Adam seems perform best as for achieving a minimum and SGD seems good at converging quickly. So, if cares about optimization speed, it is better to choose SGD.

iv. CONCLUSION

In summary, RMSprop is an extension of Adagrad that deals with its radically diminishing learning rates. It is identical to Adadelta, except that Adadelta uses the RMS of parameter updates in the numinator update rule. Adam, finally, adds bias-correction and momentum to RMSprop. Insofar, RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances. Kingma et al. [6] show that its bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser. Insofar, Adam might be the best overall choice.

Interestingly, many recent papers use vanilla SGD without momentum and a simple learning rate annealing schedule. As has been shown, SGD usually achieves to find a minimum, but it might take significantly longer than with some of the optimizers, is much more reliant on a robust initialization and annealing schedule, and may get stuck in saddle points rather than local minima. Consequently, if you care about fast convergence and train a deep or complex neural network, you should choose one of the adaptive learning rate methods.

v. REFEREMCE

[1] Richard S. Sutton. Two problems with backpropagation and other steepest-descent learning procedures for networks, 1986.

[2] Yoshua Bengio, Nicolas Boulanger Lewandowski, and Razvan Pascanu. Advances in Optimizing Recurrent Networks. 2012.

[3] Timothy Dozat. Incorporating Nesterov Momentum into Adam. ICLR Workshop, (1):2013–2016, 2016.

[4] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learningand Stochastic Optimization. Journal of Machine Learning Research, 12:2121–2159, 2011

[5] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z.Mao, Marc Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large Scale Distributed Deep Networks. NIPS 2012: Neural Information Processing Systems, pages1–11, 2012.

[6] Feng Niu, Benjamin Recht, R Christopher, and Stephen J Wright. Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. pages 1–22, 2011.