Stochastic Gradient Descent Variants and Applications

Preprint · February 2022

DOI: 10.13140/RG.2.2.12528.53767

1 author:

Mert Alagözlü
Friedrich-Alexander-University Erlangen-Nürnberg
5 PUBLICATIONS **2** CITATIONS

Gradient Descent and Stochastic Gradient Descent

Variants, applications, and more

Abstract

Our article starts defining stochastic gradient variants, its advantages, disadvantages, then continues to explain SGD based applications using Python programming language on iris dataset. It is observed that minimizing objective function for training, SGD has the lowest execution time among vanilla gradient descent and batch-gradient descent. Secondly, SGD variants are implemented on training convolutional neural networks (CNN), which leads to the fact that AdaMax has superior performance over other SGD based optimizers.

Student 1
Mert Alagözlü

Student 2
Wu Zechen

Student 3
Shi Xuetian

1 Introduction

1.1 Overview

Gradient decent approach is widely utilized in optimizing neural networks where some of the libraries such as keras offers as a blackbox optimzer. [15] Gradient descent is a common way to minimize objective function of an optimization problem. [15] Gradient of a function represents rate of increase via direction and magnitude. [1] By incrementally updating parameters in the opposite direction, the objective function reaches to a local minimum value where a learning rate η defines the number of steps to find a local minimum. [15] It might be challenging to choose an appropriate learning rate. A learning rate that is too low results in excruciatingly slow convergence, but a learning rate that is too high might stymie convergence by causing the loss function to oscillate around the minimum or even in a wide region. [4]

1.2 Gradient Descent Variants

There 3 different variations of gradient decent, which can be divided according to their trade of between accuracy and computational complexity. The differences are related to the consideration of batch sizes.

1.2.1 Vanilla gradient descent

1.2.1.1 Definition

Before delving into Stochastic gradient descent (SGD), the concept of vanilla gradient descent is beneficial to be grasped. Similarly, any variant of gradient descent approaches can be utilized to optimize functions, more specifically, they can be used to minimize the error functions of a given problem with some iterations where error function is generally written in $p2-norm$ because convex function has minimum and possible to be found. [15] [3]

Batch gradient descent is used when the batch data contain all the training samples. In other words, it applies a parameter update on each data point. Because we must compute the gradients for the whole dataset in order to execute a single update, batch gradient descent may be highly slow and is intractable for datasets that do not fit in memory. Computing the gradient vector of the loss function for the entire dataset with respect to our parameter vector across a predetermined number of epochs is one way to demonstrate vanilla gradient descent. [15] [3]

$$\theta = \theta - \eta \nabla(J(\theta)) \tag{1}$$

1.2.1.2 Pros&Cons of Vanilla Gradient Descent

The gradient is determined by the full dataset, so it can better represent the all sample, and thus more accurately face the direction of the extreme value. The gradient descent algorithm does not guarantee that the optimized function reaches the global optimal solution. Only when the loss function is a convex function, the gradient descent algorithm can guarantee the global optimal solution. Second, the gradient descent method needs to traverse all the training data every time the model parameters are updated. When M is very large, it needs to consume huge computing resources and computing time.

1.2.2 Stochastic Gradient Descent

1.2.2.1 Definition

First consider a simple supervised learning setup. Each example is a pair composed of an arbitrary input and a scalar output. We consider a loss function that measures the cost of predicting against the actual output, and we choose any function that can be parametrized by a weight vector. SGD updates the parameters for each training example. The stochastic process is determined by the instances chosen randomly each iteration. In a deployed system, the stochastic algorithm can process instances in a single pass because it does not need to remember which examples were viewed during earlier iterations. As a result, it is usually significantly faster and can also be utilized for online learning. The SGD performs frequent updates with large variance, causing the objective function to vary significantly. The variability of SGD, on the other hand, allows it to reach new and potentially superior local minima. However, as SGD continues to overshoot, convergence to the exact minimum becomes more difficult without proper learning

rate. [15] [3] [2] In other words, stochastic gradient descent updates the parameters for each training example. As a result, it is frequently significantly quicker. SGD executes frequent updates with a large variance, causing the objective function to vary significantly. As a result, SGD will continue to overshoot, complicating convergence to the precise minimum. However, as the learning rate is gradually reduced, SGD exhibits the same convergence behaviour as batch gradient descent, virtually surely converging to a local or global minimum.

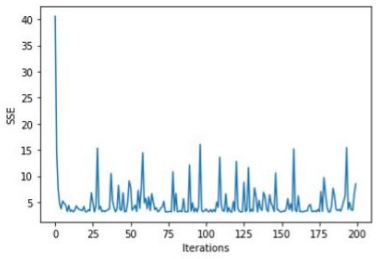

Figure 1. SGD $\eta_0 = 0.1$

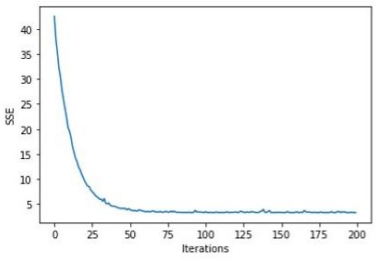

Figure 2. SGD $\eta_0 = 0.02$

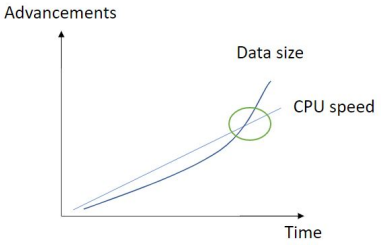

Figure 3. Threshold for SGD's Importance

$$\theta = \theta - \eta \nabla (J(\theta x_i y_i)) \tag{2}$$

The downside of SGD algorithm is that it has rapid fluctuations due to its randomness across the data points as can be seen in 1. However, as we increase the learning rate, SGD converge better with less noise as presented in 2. [15]

1.2.2.2 Recommendations for SGD

A set of recommendations for employing stochastic gradient techniques exists. Although some of these suggestions may appear basic, they are proven to be effective if not completely disregarded. Preparation of the data resolves problems such as sequential zipping in a particular order; therefore, a random shuffle is necessary. [15] [3] Stochastic gradient descent is a first-order method; therefore, it suffers when it reaches a region where the Hessian is ill-conditioned. [3] Being ill-conditioned means that the ratio between the highest singular value and the lowest singular value is large; thus, the cost function is transferred into a thin line where the minimization process becomes ineffective. Fortunately, there are several approaches, such as replacing the learning rate with a positive definite matrix, which we call the 2SGD method. [3]

For monitoring and debugging purposes, we should allocate a portion of the dataset for validation to track the error while training because it can be beneficial for us to pinpoint the exact time when the error gets stable. Since stochastic gradient descent is an iterative optimization process, the training cost must be computed regularly as well. [3] Computing the training cost and the validation error is a significant computational task since it demands extra iterations over the training and validation datasets. Another factor would be to check the gradient. When the gradients are marginally poorly computed, stochastic gradient descent typically operates slowly and leads to unstable performance. People who generally complain about finding an appropriate learning rate do not notice where exactly the problem originated. The majority of the problems come from miscalculated gradients, and the proper way to check gradients is the finite differences method (FDM). [3] Some other minor recommendations include but are not limited to; normalization, setting early stopping criteria, and adding Gaussian noise to the gradient to train a superior model. [15] Finally, experimenting with learning rates on a small dataset before the actual training might be a smart idea. Once you find a learning rate for the smaller dataset, you can apply the same value to your entire data set. [3]

1.2.2.3 Finite Difference Method

In order to check your gradient is correctly calculated, you can apply finite difference method to asses its condition. This process should be repeated for different data points. First, we pick an example, calculate the loss for the current weight, and calculate the gradient. Apply slight perturbation and verify the perturbed loss function, which can be approximated by new loss. [3]

$$J(z, w) \tag{3}$$

$$w' = w + \delta \tag{4}$$

$$\delta = -\eta \nabla_w J(z, w) \tag{5}$$

$$J'(z, w') = J(z, w) + \delta \tag{6}$$

1.2.2.4 The Convergence of SGD

The Robbins-Siegmund theorem allows for virtually certain convergence under unexpectedly moderate circumstances, even when the loss function is non-convex. [2] When the learning rates slow down too much, the variance of the estimated parameter slows down as well. When learning rates fall too fast, the expectation of the parameter estimate takes a very long time to reach the optimum. [2] [3] When the cost function's Hessian matrix (C) is strictly positive definite, which also means our cost function is at the local minimum, taking the learning rate proportional to $1/t$ yields the fastest convergence speed. On the other hand, if we set the learning rate proportional to $1/t^2$, we can expect a slower convergence rate in the final optimization process. [2]

$$xCx^T = \lambda ||x||^2 : \lambda > 0 \tag{7}$$

1.2.2.5 Pros&Cons of SGD

This algorithm optimizes not the loss function on all training data, but randomly optimizes the loss function on a sample of training data in each iteration, so that the update speed of parameters is greatly accelerated. This algorithm optimizes not the loss function on all training data, but randomly optimizes the loss function on a sample of training data in each iteration, so that the update speed of parameters is greatly accelerated.

1.2.3 Mini-batch gradient descent

Ultimately, mini-batch gradient descent combines the best of the two variants, doing an update for each mini-batch of m training samples. This diminishes the variance of parameter updates, that might also contribute to even more steady convergence. However, all parameter changes are subject to the same learning rate. If our data is scarce and our features have widely disparate frequencies, we may not wish to update them all to the same amount, but instead do a greater update for rarely occurring features.

$$\theta = \theta - \eta \nabla (J(\theta x_{(i+m)} y_{(i+m)})) \tag{8}$$

1.2.3.1 Pros&Cons of Mini-batch Gradient Descent

Through matrix operations, optimizing parameters on one Mini-batch will not cost much more time than a sample data. Second,using mini-batch can greatly reduce the number of iterations required for convergence, and at the same time can make the result of convergence closer to the effect of gradient descent. Don't have significant reduction of training time when the training dataset don't have large amount of data. Even probably cost more time than fully gradient descent, because it must pick m sample as mini-batch in each iteration, this process will take some time.

1.3 Application: SGD Adaline Using Iris Dataset

One of the other classical learning strategies where we can apply SGD. Adaline is identical to linear regression problem if you disregard the activation function. To optimize the model weight, the cost function is defined as sum of squared error function, which measures the distance between the predicted labels and the actual labels. This cost function is our objective function and must be minimized with SGD. [2] [3]

$$\Phi(z) = \begin{cases} z, & \text{if } z \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

$$z = \omega_0 x_0 + \omega_1 x_1 + ... = \sum x_i \omega_i = \boldsymbol{w}^T \boldsymbol{x} \tag{10}$$

$$J(x) = \frac{1}{2}(y^{(i)} - \Phi(z)^{(i)})^2 \tag{11}$$

$$\frac{\partial J}{\partial W_t} = -(y^{(i)} - \phi(z_t)^{(i)}) x_t^{(i)}) \tag{12}$$

$$w_{t+1} = w_t + \eta_t \nabla_w J(w_t; x_t^{(i)}, y_t^{(i)}) \tag{13}$$

The gradient of the sum of square function can be calculated with equation 12. [2] [3] Normally, if we accumulate the weights, this is considered as gradient descent rather than SGD. Rather, we update the weights at each training sample. If you look at our implementation, first, we initialize the weights as zeros. We divided the dataset into two sub-datasets as test and train with the ratio of 0.2. Using both training features and classes, our model is fitted. Then, we calculate predicted output using test features and compare the predicted classes to the actual test labels. We used iris data set to train our model and acquired almost 100% accuracy at each feature pairs using SGD as presented through figures 4 to 9 Adaline is also runned several times to demonstrate the performance difference between SGD and vanilla GD. The results regarding the same dataset are produced as 0.287 seconds and 0.997 second respectively. However, this results is stochastic and depends on different machines due to hardware differences and machine epsilons, therefore, it is possible to obtain similar but different results.

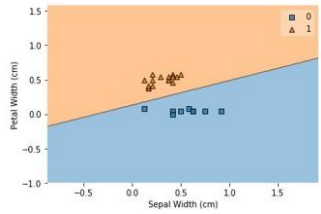

Figure 4. Adaline SGD: Accuracy 100%

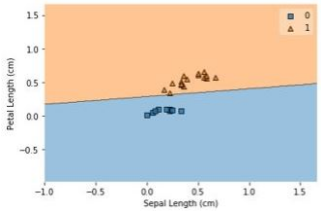

Figure 5. Adaline SGD: Accuracy 100%

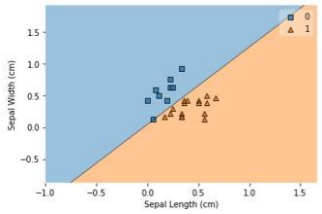

Figure 6. Adaline SGD: Accuracy 98.67%

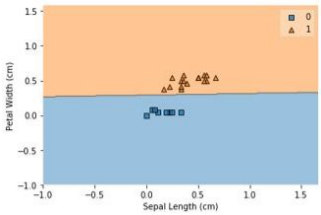

Figure 7. Adaline SGD: Accuracy 100%

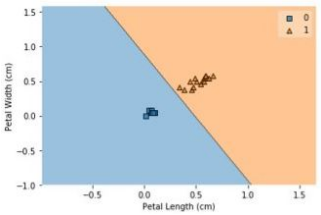

Figure 8. Adaline SGD: Accuracy 100%

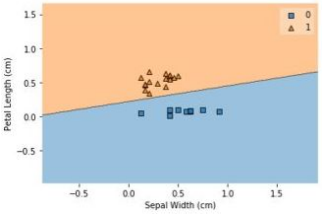

Figure 9. Adaline SGD: Accuracy 100%

1.4 Application: SGD SVM Using Iris Dataset

$$J(w) = \frac{1}{2}||\omega||_2^2 + argmax(0, 1 - y^{(i)}(\omega x^{(i)})) \tag{14}$$

$$\frac{\partial J}{\partial w_t} = \begin{cases} \omega_t, & \text{if } argmax(0, 1 - y^{(i)}\omega_t x^{(i)}) = 0 \\ \omega_t - C y^{(i)} x^{(i)}, & \text{otherwise} \end{cases} \tag{15}$$

$$\omega_{t+1} = \omega_t - \eta_t \nabla_w J(\omega_t; x_t^{(i)}, y^{(i)}) \tag{16}$$

Another implementation of SGD algorithm can be encountered in support vector machine (SVM). Our aim is to find a cost function that is eventually optimized with SGD. Our cost function is represented as J. As C gets higher, we have small margins. Like previous steps, we first need to calculate the gradient and move along the opposite direction at each training sample rather than the whole training set while scaling with the learning rate until the convergence occurs. [2] One weakness of this methodology might be that the decision boundary cannot be generalized with linearly representable solution even though linear decision boundary is sufficient for this dataset. As you can see, for all possible 2-dimensional feature space, we achieved around 90 percent accuracy with relatively quick convergence time to gradient descent. [2] [3]

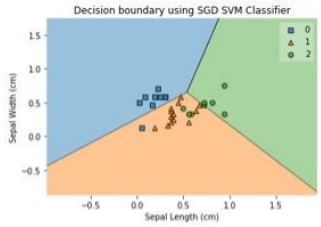

Figure 10. SGD SVM: Accuracy 86.67%

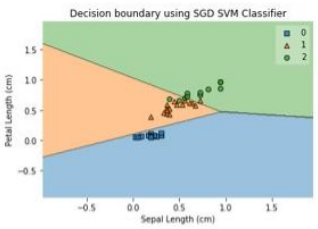

Figure 11. SGD SVM: Accuracy 89.47%

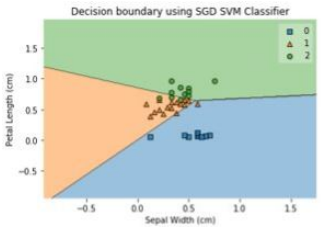

Figure 12. SGD SVM: Accuracy 89.47%

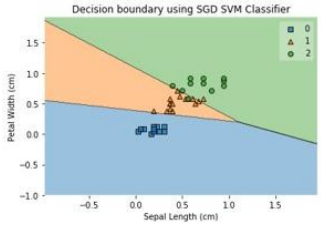

Figure 13. SGD SVM: Accuracy 94.74%

Figure 14. SGD SVM: Accuracy 92.11%

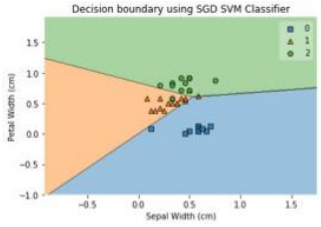

Figure 15. SGD SVM: Accuracy 92.11%

1.5 Application: SGD Perceptron using Iris Dataset

The perceptron is an algorithm for supervised learning of binary classifiers, It is a type of linear classifier [8]. The mathematical expression of gradient algorithms for perceptron is represented below. [2]

$$\begin{aligned} &Q_{\text{perceptron}} = \max\left\{0, -yw^{\top}\Phi(x)\right\} \\ &\Phi(x) \in \mathbb{R}^d, y = \pm 1 \end{aligned} \qquad w \leftarrow w + \gamma_t \begin{cases} y_t \Phi(x_t) & \text{if } y_t w^{\top}\Phi(x_t) \leq 0 \\ 0 & \text{otherwise} \end{cases} \tag{17}$$

Here is the results of comparison for three different gradient descend method:

	accuracy	running time
Fully GD	92.67%	0.03s
SGD	93.33%	0.02s
Mini-batch SGD	92.67%	0.06s

Table 1. Execution times and accuracies of different methods

From the result the accuracy of three method is quite close, but the running time is different, SGD spend the least time, then is Fully GD, Mini-batch SGD spent the most time. It should be noted that because of the computer architecture or system capabilities, the running time might differ from machine to machine.

2 Application: Stochastic Gradient Descent Variants on Convolutional Neural Networks

We used SGD based optimizers in the context of Convolutional neural network since it is well-known fact that the selecting learning rate is problematic. [9] Therefore, we introduced other variants. As mentioned before, the trainable dataset *Iris dataset* has 150 samples. It has 3 class labels and 4 features.

Then we train the model, after 3000 epochs the training and validation loss plot is showed in figure 16: From the plot, we can see that the model converge after about 1500 epochs. At this time, training loss is 0.1388 and validation loss is 0.1391. Additionally, training accuracy is 94% and validation accuracy is 94%. After that, the model will be overfitting. One drawback of SGD method is that its update direction is completely dependent on the gradient calculated by the current batch, so it is very unstable. However, momentum algorithm will observe the historical gradient v_{t-1} by adding a fraction γ. [16] If the current gradient direction is consistent with the historical gradient, the gradient in this direction will be enhanced. In contrast, if the current gradient is inconsistent with the historical gradient direction, the gradient will be weakened.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta),\ where\ \theta = \theta - v_t \tag{18}$$

As showed in the figure 17, the model converge much more quickly than SGD.

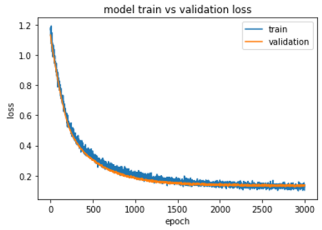

Figure 16. SGD

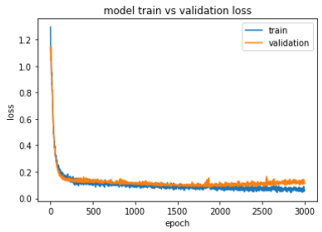

Figure 17. SGD Momentum

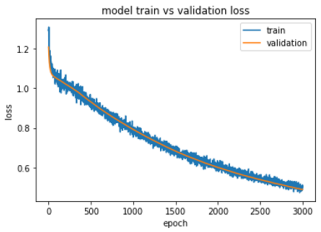

Figure 18. Adagrad

Another method is called nesterov accelerated gradient. Compared with Momentum, this method take the derivative at $\theta - \gamma v$ instead of θ. It first makes a big jump in the direction of the previous accumulated gradient, measures the gradient and then makes a correction. [15] So nesterov accelerated gradient can prevent us from going too fast and results in increased responsiveness, which has significantly increased the performance of RNNs. And the training and validation loss plot is similar with SGD Momentum. Adagrad algorithm can automatically adjust the learning rate during training. It adopts large α update for parameters with low frequency, and adopts smaller α update for parameters with high frequency. [7] So it is great for handling sparse data. The training and validation loss plot is showed in figure 18: We can see this plot is very different than the previous. It converge very slow. And after 3000 epochs, training accuracy is only 87% and validation aaccuracy is 78%. Adadelta optimizer is an extension of Adagrad that attempts to reduce its aggressive, monotonically reduced learning rate. It does not accumulate all past square gradients, but rather limits the window of accumulated past gradients to a certain size. RMSprop only calculates the corresponding average value, so it can ease the problem of Adagrad algorithm learning rate decline quickly. It is suitable for handling non-stationary targets and especially works well for RNN. [15] Adam is another method of adaptive learning rate. The learning rate of each parameter is dynamically adjusted by first- and second-order moment estimation of the gradient. The main advantage is that after bias correction, each iteration learning rate has a certain range, which makes the parameters relatively stable. [10] So it preserves an exponentially decaying average of previous gradients in addition to an exponentially decaying average of past squared gradients. [11] [15] Nadam is similar to Adam but with the Nesterov momentum term. It has a stronger constraint on the learning rate and a more direct effect on the updating of the gradient. In general, where you want to use drive volume RMSprop, or Adam, most can use Nadam for better results. [6] [15]

2.1 Which optimizer is best?

In this application, we train CNN in Iris datasset, and use different gradient descend methods to optimize. Here is the results:

	training accuracy	validation accuracy	epochs	running time(3000 epochs)
SGD	94%	94%	3000	126s
SGD Momentum	93%	94%	200	118s
Nesterov accelerated gradient	94%	96%	100	113s
Adagrad	87%	78%	3000	143s
Adadelta	53%	56%	3000	113s
RMSprop	96%	98%	150	142s
Adam	94%	98%	300	142s
AdaMax	93%	94%	500	110s
Nadam	95%	96%	300	113s

Table 2. Comparison of different variants on CNN

From table 2, all have similar results. For sparse data, the adaptive optimization method of learning rate should be used as far as possible. SGD usually takes longer to train, but results are more reliable in the case of good initialization and learning-rate scheduling schemes. Moreover, if you care about faster convergence and need to

train deeper and more complex networks, it is recommended to use the learning rate adaptive optimization method. Adadelta, RMSprop and Adam are relatively similar algorithms, which perform almost the same in similar cases. Insofar, Adam might be the best overall choice. [15] However, according to the table 2 AdaMax provides faster execution time than other variants.

3 Parallelized SGD

Parallelized SGD can be integrated into MapReduce where we can achieve even faster results. Earlier attempts to utilize distributed gradient calculations locally on each computer node that maintains portions of the data, followed by gradient aggregation to achieve a global update step. Each computer node must be synchronous and transfer information among them. In a simpler approach, parallelized SGD runs the SGD algorithm in each processor with a fixed learning rate on the local MapReduce"d" dataset over some iterations. Additionally, while running SGD on each processor where the MapReduce"d" data is located, we also run a master routine to record solutions coming from each computer node. On each computer node, we get updated weights, take their average, and return it at the end of the iterations. This process does not require constant communication between computer nodes, resulting in much faster convergence, preferably in the MapReduce framework. [18]

$$i \in 1, ..., k,\ v_i = SGD(\Phi(z)^{(i)}, T, \eta, \omega_0),\ v = \frac{1}{k}\sum_{i=1}^{k} v_i \tag{19}$$

3.1 Optimize Parallelized and Distributed SGD

There are several algorithms to optimize the parallelized SGD.Hogwild enables SGD updates on parallel processors. Processors are allowed to use a shared memory without forcing weights to be pre-determined. However, since processors share a memory, the bandwidth limitation might be a problem in overly large datasets. [14] [18] Downpour SGD is an asynchronous variant of SGD, each computer node or processor responsible storing and updating weights, however, processors do not communicate with each other, which might result in divergence. [5] [18] Delay-tolerant algorithms for SGD uses AdaGrad, which adapts to past gradients and update delays accordingly. [13] [18] Tensorflow Google's initiative that enables users to train large scale machine learning models with minor code change. For example, you can use distribute Strategy API to train your model across many CPUs, GPUs and TPUs, which might offer fast implementation if you aim to parallelize your training dataset. [12] Elastic Averaging ties weights with an elastic force, which lead to fluctuation and prospective better local minimums. [17] [18]

4 Conclusion

From our observation of application result and analysis of mathematical properties, we observe that SGD as gradient method can reduce a lot of training time comparing other variations of gradient descent, especially when training iris dataset. Two possible outcomes is deducted. SGD has the shortest execution time among vanilla gradient descent and batch-gradient descent when the goal function for training is minimised. Second, SGD variations are used to train convolutional neural networks (CNN), resulting in AdaMax outperforming other SGD-based optimizers.

References

[1] D. Bachman. *Advanced calculus demystified*. McGraw Hill Professional, 2011.

[2] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[3] L. Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer, 2012.

[4] N. Buduma and N. Locascio. *Fundamentals of deep learning*. O'Reilly Media, 2017.

[5] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, et al. Large scale distributed deep networks. 2012.

[6] E. Dogo, O. Afolabi, N. Nwulu, B. Twala, and C. Aigbavboa. A comparative analysis of gradient descent-based optimization algorithms on convolutional neural networks. In *2018 International Conference on Computational Techniques, Electronics and Mechanical Systems (CTEMS)*, pages 92–99. IEEE, 2018.

[7] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[8] R. E. Freund, Y.; Schapire. Large margin classification using the perceptron algorithm. In *Machine Learning*, page 277–296. 1999.

[9] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[10] D. Kingma and J. Ba. Adam: A method for stochastic optimization, Apr 2015.

[11] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2017.

[12] G. LLC. Distributed training with tensorflow. `https://www.tensorflow.org/guide/distributed_training`. Accessed: 2022-01-02.

[13] B. McMahan and M. Streeter. Delay-tolerant algorithms for asynchronous distributed online learning. *Advances in Neural Information Processing Systems*, 27:2915–2923, 2014.

[14] F. Niu, B. Recht, C. Ré, and S. J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *arXiv preprint arXiv:1106.5730*, 2011.

[15] S. Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[16] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.

[17] S. Zhang, A. Choromanska, and Y. LeCun. Deep learning with elastic averaging sgd. *arXiv preprint arXiv:1412.6651*, 2014.

[18] M. Zinkevich, M. Weimer, A. J. Smola, and L. Li. Parallelized stochastic gradient descent. In *NIPS*, volume 4, page 4. Citeseer, 2010.

A Python code of SGD Adaline Using Iris Dataset

```
from mlxtend.data import iris_data
from mlxtend.plotting import plot_decision_regions
from mlxtend.classifier import Adaline
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn.model_selection import train_test_split as trtesp
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report
```

```
from sklearn.preprocessing import MinMaxScaler
import time
# Loading Data

"""
please install pip install mlxtend for direct use of iris dataset via an API
"""

#print(X)

start = time.time()
def AdalineSGD(X,y,epochs,eta):
    X_train, X_test, y_train, y_test = trtesp(X, y, test_size=0.25, random_state=25)

    ada = Adaline(epochs=epochs,
                  eta=eta,
                  minibatches=len(y),
                  random_seed=1,
                  print_progress=3)

    ada.fit(X_train, y_train)
    run_time = time.time() - start
    print(run_time)
    plot_decision_regions(X_train, y_train, clf=ada)
    plt.title('')
    plt.show()


#    y_predict = ada.predict(X_test)

    plot_decision_regions(X_test, y_test, clf=ada)
    plt.xlabel("Sepal Length (cm)")
    plt.ylabel("Sepal Width (cm)")
#    plt.xlabel("Sepal Length (cm)")
#    plt.ylabel("Petal Length (cm)")
#    plt.xlabel("Sepal Length (cm)")
#    plt.ylabel("Petal Width (cm)")
#    plt.xlabel("Sepal Width (cm)")
#    plt.ylabel("Petal Length (cm)")
#    plt.xlabel("Sepal Width (cm)")
#    plt.ylabel("Petal Width (cm)")
#    plt.xlabel("Petal Length (cm)")
#    plt.ylabel("Petal Width (cm)")

#    print('Adaline with SGD Classifier:', ada.score(X_train, y_train)*100,'%')
#    cmatrix = confusion_matrix(y_test, y_predict)
#    print("\nClassification Report")
#    report = classification_report(y_test, y_predict)
#    print(report)
#

X, y = iris_data() # X sepal measurements where y is labels

#X = sk.preprocessing.normalize(X)
X = MinMaxScaler().fit_transform(X)
```

```
X1 = X[:, [0, 1]]# sepal length and sepal width in cm
X2 = X[:, [0, 2]]#  sepal length and petal length in cm
X3 = X[:, [0, 3]]# sepal length and petal width in cm
X4 = X[:, [1, 2]]# sepal width and petal length in cm
X5 = X[:, [1, 3]]# sepal width and petal width in cm
X6 = X[:, [2, 3]]# petal length and petal width in cm

X1 = X1[0:100]
y = y[0:100]

#start
#   for i in range(100):
AdalineSGD(X1,y,200,0.02)

#print(run_time)


#SSR=0
#for i in range(len(y_predict)):
#
#    SSR += (y_predict[i] - y_test[i])**2
#plt.plot(range(len(ada.cost_)), ada.cost_)
#plt.xlabel('Iterations')
#plt.ylabel('SSE')
#plt.show()
```

B SGD SVM Using Iris Dataset

```
#pip install mlxtend
from mlxtend.data import iris_data
from mlxtend.plotting import plot_decision_regions
from mlxtend.classifier import Adaline
import matplotlib.pyplot as plt
import numpy as np
import sklearn as sk
from sklearn.model_selection import train_test_split as trtesp
from sklearn.metrics import accuracy_score
from sklearn.utils import shuffle
#from sklearn.linear_model import SGDOneClassSVM
from sklearn import linear_model
from sklearn.linear_model import SGDClassifier

from mlxtend.data import iris_data
from mlxtend.plotting import plot_decision_regions
from mlxtend.classifier import Adaline
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report
import time
```

```
X, y = iris_data()
#print(y)

#print(X)
#X = sk.preprocessing.normalize(X)
X = MinMaxScaler().fit_transform(X)

#X = X- X.mean()/X.std()
#print(X)

"""
0. sepal length in cm
1. sepal width in cm
2. petal length in cm
3. petal width in cm

class:
0 Iris Setosa
1 Iris Versicolour
2 Iris Virginica
"""



def SVMSGD(X,Y,alpha,max_iter):

    X_train, X_test, y_train, y_test = trtesp(X, y, test_size=0.4, random_state=25)

    SGDSVM = SGDClassifier(loss="hinge", alpha=alpha, max_iter=max_iter) # Hinge represent linear model of
    SGDSVM.fit(X_train, y_train)

    SGDSVM.fit(X_test,y_test)
#    SGDSVM.partial_fit(X_test,y_test, classes=np.unique(y_test))

    y_predict = SGDSVM.predict(X_test)


#    print("Confusion Matrix")
#    cmatrix = confusion_matrix(y_test, y_predict)
#    print(cmatrix)
    # Classification Report
#    print("\nClassification Report")
#    report = classification_report(y_test, y_predict)
#    print(report)


#    accuracy = accuracy_score(y_test, y_predict)
#    print('SVM with SGD Classifier: {:.2f}%'.format(accuracy*100))


#    plot_decision_regions(X_test, y_test, clf = SGDSVM, legend = 1)
#    plt.title("Decision boundary using SGD SVM Classifier")
#    plt.xlabel("Sepal Length (cm)")
#    plt.ylabel("Sepal Width (cm)")
#    plt.xlabel("Sepal Length (cm)")
```

```
#    plt.ylabel("Petal Length (cm)")
#    plt.xlabel("Sepal Length (cm)")
#    plt.ylabel("Petal Width (cm)")
#    plt.xlabel("Sepal Width (cm)")
#    plt.ylabel("Petal Length (cm)")
#    plt.xlabel("Sepal Width (cm)")
#    plt.ylabel("Petal Width (cm)")
#    plt.xlabel("Petal Length (cm)")
#    plt.ylabel("Petal Width (cm)")


    SSR=0
    for i in range(len(y_predict)):
        SSR += (y_predict[i] - y_test[i])**2


X1 = X[:, [0, 1]]# sepal length and sepal width in cm
X2 = X[:, [0, 2]]#  sepal length and petal length in cm
X3 = X[:, [0, 3]]# sepal length and petal width in cm
X4 = X[:, [1, 2]]# sepal width and petal length in cm
X5 = X[:, [1, 3]]# sepal width and petal width in cm
X6 = X[:, [2, 3]]# petal length and petal width in cm

X = X[0:150]
y = y[0:150]



start = time.time()
for i in range(100):
    SVMSGD(X1,y,0.0001,2000000)
    run_time = time.time() - start
print(run_time/100) # Execution time

"""

#plt.plot(range(len(SGDSVM.scores)), SGDSVM.scores)
#plt.xlabel('Iterations')
#plt.ylabel('SSE')
#plt.show()

        """
```

C SGD Perceptron using Iris Dataset

```
import numpy as np
import pandas as pd
import random
import time

def FGD_train(X, y, alpha=0.0001, maxIter=1000, theta_old=None):
    '''
    Fully GD training linear regression
```

```
    incoming:
        X       :  training data
        y       :  label
        alpha   :  learning rate
        maxIter :  Number of iterations

    return:
        theta : Weight parameter
    '''
    # Initialize weight parameters
    theta = np.ones(shape=(X.shape[1],))
    if not theta_old is None:
        theta = theta_old.copy()

    for i in range(maxIter):
        # predict
        y_pred = np.sum(X * theta, axis=1)
        # Gradient from all data
        gradient = np.average((y - y_pred).reshape(-1, 1) * X, axis=0)
        # update learning rate
        theta += alpha * gradient
    return theta


def SGD_train(X, y, alpha=0.0001, maxIter=1000, theta_old=None):
    '''
    SGD training linear regression
    incoming:
        X       :  training data
        y       :  label
        alpha   :  learning rate
        maxIter :  Number of iterations

    return:
        theta : weight parameter
    '''
    # Initialize weight parameters
    theta = np.ones(shape=(X.shape[1],))
    if not theta_old is None:
        theta = theta_old.copy()
    # amount of data
    data_length = X.shape[0]
    for i in range(maxIter):
        # randomly select a data
        index = np.random.randint(0, data_length)
        # predict
        y_pred = np.sum(X[index, :] * theta)
        # Gradient from the slected data
        gradient = (y[index] - y_pred) * X[index, :]
        # update learning rate
        theta += alpha * gradient
    return theta


def MBGD_train(X, y, alpha=0.0001, maxIter=1000, batch_size=10, theta_old=None):
```

```
    '''
    Mini-batch GD training linear regression
    incoming:
        X       :  training data
        y       :  label
        alpha   :  learning rate
        maxIter :  Number of iterations
        batch_size :  The number of data fed in each round

    return:
        theta : weight parameter
    '''
    # Initialize weight parameters
    theta = np.ones(shape=(X.shape[1],))

    if not theta_old is None:
        theta = theta_old.copy()

    # collection of all data
    all_data = np.concatenate([X, y.reshape(-1, 1)], axis=1)
    for i in range(maxIter):
        # Select batch_size items from all data
        X_batch_size = np.array(random.choices(all_data, k=batch_size))

        # reassign X, y
        X_new = X_batch_size[:, :-1]
        y_new = X_batch_size[:, -1]

        # update theta
        theta = FGD_train(X_new, y_new, alpha=0.0001, maxIter=1, theta_old=theta)
    return theta



def GD_predict(X, theta):
    '''
    function for prediction
    Incoming:
         X : data
         theta : weight
     return:
         y_pred: prediction vector
    '''
    y_pred = np.sum(theta * X, axis=1)
    y_pred = (y_pred + 0.5).astype(int)
    return y_pred


def calc_accuracy(y, y_pred):
    '''
    Calculate accuracy
    Incoming:
         y : label
         y_pred : predicted value
     return:
```

```
        accuracy : accuracy rate
    '''
    return np.average(y == y_pred)*100

# read data
iris_raw_data = pd.read_csv('./iris.data', names  =['sepal length', 'sepal width', 'petal length', 'petal w

# Mapping three types to integers
Iris_dir = {'Iris-setosa': 1, 'Iris-versicolor': 2, 'Iris-virginica': 3}
iris_raw_data['class'] = iris_raw_data['class'].apply(lambda x:Iris_dir[x])


# training data X
iris_data = iris_raw_data.values[:, :-1]

# label y
y = iris_raw_data.values[:, -1]

# training with Fully GD
start = time.time()
theta_MGD = FGD_train(iris_data, y)
run_time = time.time() - start
y_pred_MGD = GD_predict(iris_data, theta_MGD)
print("accuracy of fully GD training after 1000 iterations {:.2f}% running time{:.2f}s".format(calc_accurac

# training with SGD
start = time.time()
theta_SGD = SGD_train(iris_data, y)
run_time = time.time() - start
y_pred_SGD = GD_predict(iris_data, theta_SGD)
print("accuracy of SGD training after 1000 iterations {:.2f}% running time{:.2f}s".format(calc_accuracy(y,

# training with MBGD
start = time.time()
theta_MBGD = MBGD_train(iris_data, y)
run_time = time.time() - start
y_pred_MBGD = GD_predict(iris_data, theta_MBGD)
print("accuracy of mini-batch SGD training after 1000 iterations {:.2f}% running time{:.2f}s".format(calc_a
```

D CNN using iris Dataset

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.preprocessing import LabelEncoder
import tensorflow as tf
import keras
from keras.models import Sequential
from keras.layers import Dense, Flatten, Conv1D, MaxPool1D, Dropout
from keras.utils import np_utils
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from mpl_toolkits.mplot3d import Axes3D
import time
```

```
# load iris data
iris = load_iris()
x_reduced = PCA(n_components=3).fit_transform(iris.data)

#PCA
fig = plt.figure()
ax = Axes3D(fig)
ax.set_title('Iris Dataset By PCA',size=14)
ax.scatter(x_reduced[:,0],x_reduced[:,1],x_reduced[:,2],c=Species)
ax.set_xlabel('First eigenvector')
ax.set_ylabel('Secnd eigenvector')
ax.set_zlabel('Third eigenvector')
ax.w_xaxis.set_ticklabels(())
ax.w_yaxis.set_ticklabels(())
ax.w_zaxis.set_ticklabels(())

Species = iris.target
x_train, x_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.33, random_state=1)

# The known number of output classes.
num_classes = 3

#  label encoding
encoder = LabelEncoder()
y_train = encoder.fit_transform(y_train)
y_test = encoder.fit_transform(y_test)

# one hot encoding
y_train = np_utils.to_categorical(y_train, num_classes)
y_test = np_utils.to_categorical(y_test, num_classes)

# reshape 2D to 3D
x_train = x_train.reshape(100, 4, 1)
x_test = x_test.reshape(50, 4, 1)

# build CNN model
model = Sequential()
model.add(Conv1D(64, 2, input_shape=(4, 1), activation='relu'))  # convolation
model.add(MaxPool1D(pool_size=2))  # pooling
model.add(Flatten())  # flatten
model.add(Dense(128, activation='relu'))  # fc
model.add(Dropout(0.3))  # dropout
model.add(Dense(num_classes, activation='softmax'))

# model compile
start = time.time()
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=tf.keras.optimizers.Nadam(), # can change it
              metrics=['accuracy'])
# model.summary()

batch_size = 128
epochs = 3000
model = model.fit(x_train, y_train,
                  batch_size=batch_size,
```

```
                  epochs=epochs,
                  verbose=2,
                  validation_data=(x_test, y_test))
run_time = time.time() - start
print("Total running time:",run_time)

# draw loss
plt.plot(model.history['loss'])
plt.plot(model.history['val_loss'])
plt.title('model train vs validation loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```