Performance Evaluation of different Optimizers on LeNet-5

ECE 579 Spring 2021 Final Project

Zihao Ding
Rutgers University
New Brunswick. NJ. USA
zd75@scarletmail.rutgers.edu

ABSTRACT

This paper compares the performance of different types of neural network optimizers on a LeNet-5 network using MNIST database. The optimizer evaluated in this paper are SGD, AdaGrad, RMSprop and Adam.

KEYWORDS

LeNet-5, SGD, AdaGrad, RMSprop, Adam, MNIST

1 Introduction

Deep learning models contain a large number of parameters, in order to correct these parameters, many optimization algorithms have been proposed. In this paper, different types of optimizers, including SGD, AdaGrad, RMSprop, Adam, were tested on a LeNet-5 network with MNIST database as test and train input. Their performance is compared and analyzed.

The organization of this paper is as follows. Section 2 presents the basic terminologies used in this paper. Section 3 introduces technical details about the experiment. Section 4 shows the experimental result based on MNIST database with analysis. Section 5 is the conclusion of this paper. All code used in the experiment is attached in the Appendix.

2 Terminology

In this part, I will briefly explain all the terminologies used in this paper, please refer to the source for more detailed explanation.

2.1 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is one of the most basic iterative optimization algorithms for neural networks at the moment. It is an “upgraded” algorithm based on the basic Gradient Descent. Where the Gradient Descent algorithm is the algorithm used to find the value of “x” where the corresponding “y” is minimum. It is an is an iterative algorithm, which starts from a point on a function and travels down the slope in steps until it reaches the minimum point. Stochastic Gradient Descent is similar to Gradient Descent, its basic idea is to approximating the true gradient computed over the entire data set with the gradient computed over a random, small-batch subset. Compared to the basic Gradient Descent, SGD randomly picks one data point from the whole data set at each iteration to reduce the computations. Formula (1) presents the basic of SGD.[1]

$$\theta_{i\_new} = \theta_i - \eta \nabla_{\theta_i} J(\theta) \quad (1)$$

SGD has some drawbacks. First, it is easy to fall into local optimum for non-convex error functions. Second, it is difficult to choose the appropriate learning rate for SGD. A learning rate that is too small may lead to very slow convergence, while a learning rate that is too large will obstruct the convergence and cause the weights to oscillate around the optimal solution, and in some cases may even cause divergence.

2.2 Adaptive Gradient Algorithm (AdaGrad)

AdaGrad, also known as the adaptive gradient algorithm, is another derivative of the standard gradient descent algorithm. AdaGrad is also an upgrade to the SGD. The main idea of AdaGrad is to use different learning rates in each dimension, thus speeding up the convergence of the function. Formula (2) presents the basic of AdaGrad. [2]

$$\theta_{i,t+1} = \theta_{i,t} - \frac{\eta}{\sqrt{G_{i,t} + \epsilon}} \nabla_{\theta_{i,t}} J(\theta) \quad (2)$$

Notice the difference between (1) and (2). In short, after setting the global learning rate, each time the global learning rate is passed, the global learning rate is divided by the square root of the sum of the squares of the historical gradients parameter by parameter, making the learning rate of each parameter to be different. Adagrad is designed to cope with large-scale problems where the number of dimensions makes it impractical to manually choose a different learning rate for each dimension.

2.3 Root mean square prop (RMSprop)

RMSprop is proposed by Geoffrey Hinton of the University of Toronto in an online class. [3] It is an unpublished optimization algorithm for neural network. RMSprop is an improvement of the AdaGrad algorithm. It divides the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight. Given that the neural networks are all non-convex, RMSProp gives better results in non-convex conditions. Compared to AdaGrad's history gradient, RMSProp adds a decay factor to control how much history information is obtained. Formula (3) presents the basic of RMSprop. [3]

(3) $$MeanSquare(w,t) = 0.9\ MeanSquare(w,\ t{-}1) + 0.1\left(\partial E/_{\partial w}(t)\right)^2$$

2.4 Adam

Adam's algorithm is another upgrade on the optimization algorithm, it can be seen as the algorithm with modified RMSProp and Momentum. As the previous algorithms, Adam is also an extension to the SGD. It was presented by Diederik Kingma in 2015 [4]. Adam combines the advantages of previous algorithms. From AdaGrad, it maintains a parameter learning rate that improves the performance on sparse gradient problems; and from RMSProp, it makes use of the average of the second moments of the gradients, and the momentum is directly incorporated into the gradient first-order moment estimation. In general, Adam combines the best feature of both AdaGrad and RMSProp and provides an algorithm that can handle noise and sparse gradients. This is also the reason why it becomes one of the most popular optimizers right now.

2.5 LeNet-5

LeNet-5 is a common model of convolutional neural network structure proposed by Yann LeCun et al. in 1989. Its architecture is showed in the Figure below [5].

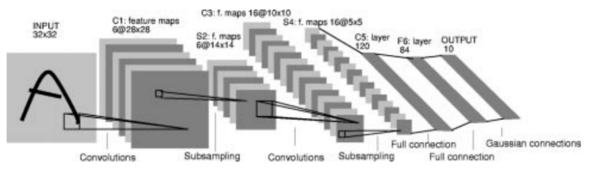


Figure 1: The basic Architecture of LeNet-5. This image is from [5]

The LeNet-5 has 7 layers: three convolutional layers, two subsampling layers, one fully connected layer and output layer. As showed in the Figure, the first layer (C1) is a convolutional layer which has 6 feature planes of 28x28 pixels. The second layer (S2) is the subsampling layer where the planes from previous layer are reduced into the size of 14x14. The third layer is another convolution layer (C3) which extends the number of feature maps from 6 to 16 and size is 10x10. The subsampling layer after (S4) works the same as (S2), where it reduces the size of those 16 feature maps to half of their size (5x5). The last convolutional layer (C5) has 120 feature planes, and it performs same function as the fully connected layer. The next fully connected layer (F6) has 84 unites connected to the previous layer (C5). The final layer is the output layer with 10 neurons.

2.6 MNIST database

The MNIST database (Modified National Institute of Standards and Technology database) is a well-known database contains large number of handwritten digits, it is commonly used for training various image processing systems. The MNIST database contains 60000 training samples and 10000 test samples.

Figure 2: Sample image in MNIST database [6]

All the images in the database have been size normalized to 28x28 pixels and also centered [6].

3 Experiment

After understanding all the terminologies above, the experiment is simple. In the experiment, a neural network based on LeNet-5 structure was created, using MNIST database as testing and training dataset. Then, different optimizers were implemented during the training process in order to see the changes in loss during training process and the change in final accuracy and runtime performance.

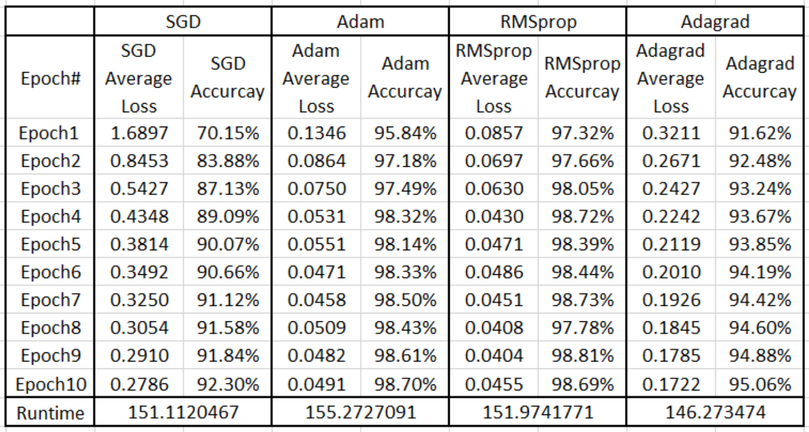
Due to the limited computing resource and time, all optimizers were tested with 10 epochs, with batch size of 128, and learning rate all set to 0.001 (since 0.001 is also the suggest learning rate for Adam).

All experiments were performed on a 3.2-GHz Intel Core i7-8700 CPU with 16 GB RAM running Windows 10 Pro operating system.

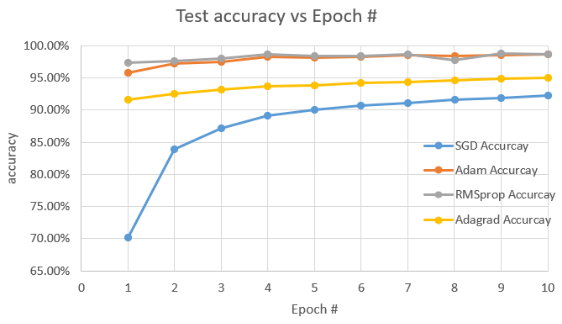
4 Results

The experiment result is presented in Figure 3 and Figure 4. Before I going to present the result, there is one thing I do want to mention, which is either to use Epoch, or Batch or Iteration as the sampling reference. This is an interesting question, when using Epoch, it can ensure all samples has been accessed with the same frequency, but for the dataset that has different size, when using the same batch size, the larger dataset will for sure has higher sampling rate than the smaller dataset because its larger, took more batches to cover all data. I tried to look for an answer but end with even more ways to determine the results which are: Loss/Epoch, Loss/iteration, Loss/total image seen (iteration* batch size), and Loss/time. I am not sure about I should either use Epoch Loss or Batch Loss as the tool for performance comparison thus I have implemented both graph in this part.

The experiment had been run several times, and the initial output of run 0, is saved in the screenshot folder for future reference. The output1.txt, output2.txt, output3.txt are results from test run 1, 2, and 3. Due to the limitation of time, these test results have not yet been plotted.



	SGD		Adam		RMSprop		Adagrad	
Epoch#	SGD Average Loss	SGD Accuracy	Adam Average Loss	Adam Accuracy	RMSprop Average Loss	RMSprop Accuracy	Adagrad Average Loss	Adagrad Accuracy
Epoch1	1.6897	70.15%	0.1346	95.84%	0.0857	97.32%	0.3211	91.62%
Epoch2	0.8453	83.88%	0.0864	97.18%	0.0697	97.66%	0.2671	92.48%
Epoch3	0.5427	87.13%	0.0750	97.49%	0.0630	98.05%	0.2427	93.24%
Epoch4	0.4348	89.09%	0.0531	98.32%	0.0430	98.72%	0.2242	93.67%
Epoch5	0.3814	90.07%	0.0551	98.14%	0.0471	98.39%	0.2119	93.85%
Epoch6	0.3492	90.66%	0.0471	98.33%	0.0486	98.44%	0.2010	94.19%
Epoch7	0.3250	91.12%	0.0458	98.50%	0.0451	98.73%	0.1926	94.42%
Epoch8	0.3054	91.58%	0.0509	98.43%	0.0408	97.78%	0.1845	94.60%
Epoch9	0.2910	91.84%	0.0482	98.61%	0.0404	98.81%	0.1785	94.88%
Epoch10	0.2786	92.30%	0.0491	98.70%	0.0455	98.69%	0.1722	95.06%
Runtime	151.1120467		155.2727091		151.9741771		146.273474	



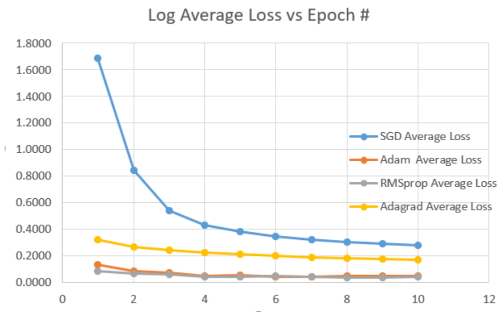


Fig 3. Experiment Result of different optimizers. Data collected via # of Epochs.

The Figure 3 presents the result which uses the Epoch as sampling reference. From the Accuracy result, it is clear that Adam out performs all other optimizers, followed by RMSprop, then Adagrad. SGD performs the worst as expected.

Figure 4 shows the result with iterations as sampling rate. It is clear that when using iterations as the sampling rate, more details can be observed than the Epoch's result. From the log axis plot, it is clear how each optimizer vibrates around the minimum loss. The experiment could have better and detailer result if run with more Epochs or smaller batch size. It is surprise to see even though Adam outperforms the rest, but it still got very similar performance with RMSprop in this case. Where the other two, the SGD and Adagrad have similar performance. Due to the size of graph, if you would like to view the full graph, please refer to the attached excel file.

5 Conclusion

In this paper, I compared the actual performance between the current popular optimizers, Adam, RMSprop, Adagrad and SGD. Since Adam is an upgrade from RMSprop, RMSprop is an upgrade from Adagrad, and Adagrad is a better version of SGD, their actual performance also follows this pattern, where Adam outperforms all other optimizers, followed by RMSprop, then is Adagrad, with SGD as the worst performance. No wonder why Adam is now the most popular optimizers.

REFERENCES

[1] Robbins, H.. "A Stochastic Approximation Method." Annals of Mathematical Statistics 22 (2007): 400-407.

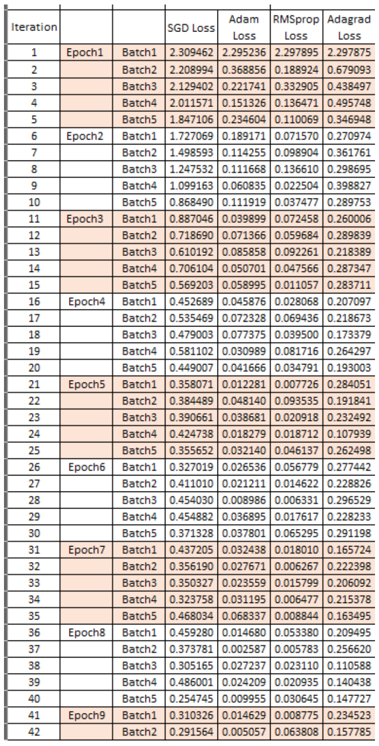
[2] John Duchi , Elad Hazan, Yoram Singer "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization" Journal of Machine Learning Research. July 11, 2011

[3] Geoffrey Hinton Neural Networks for machine learning online course. https://www.coursera.org/learn/neural-networks/home/welcome

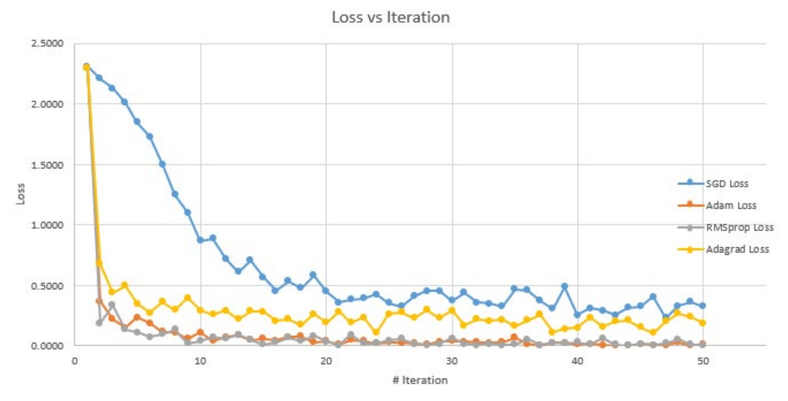
[4] Diederik P. Kingma, , and Jimmy Ba. "Adam: A Method for Stochastic Optimization." (2017). [2] Clifford A. Shaffer. Data Structures and Algorithm Analysis in Java. Third Edition. Courier Corporation. 2011.

[5] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791.

[6] "THE MNIST DATABASE of handwritten digits". Yann LeCun, Courant Institute, NYU Corinna Cortes, Google Labs, New York Christopher J.C. Burges, Microsoft Research, Redmond.



Iteration			SGD Loss	Adam Loss	RMSprop Loss	Adagrad Loss
1	Epoch1	Batch1	2.309462	2.295236	2.297895	2.297875
2		Batch2	2.208994	0.368856	0.188924	0.679093
3		Batch3	2.129402	0.221741	0.332905	0.438497
4		Batch4	2.011571	0.151326	0.136471	0.495748
5		Batch5	1.847106	0.234604	0.110069	0.346948
6	Epoch2	Batch1	1.727069	0.189171	0.071570	0.270974
7		Batch2	1.498593	0.114255	0.098904	0.361761
8		Batch3	1.247532	0.111668	0.136610	0.298695
9		Batch4	1.099163	0.060835	0.022504	0.398827
10		Batch5	0.868490	0.111919	0.037477	0.289753
11	Epoch3	Batch1	0.887046	0.039899	0.072458	0.260006
12		Batch2	0.718690	0.071366	0.059684	0.289839
13		Batch3	0.610192	0.085858	0.092261	0.218389
14		Batch4	0.706104	0.050701	0.047566	0.287347
15		Batch5	0.569203	0.058995	0.011057	0.283711
16	Epoch4	Batch1	0.452689	0.045876	0.028068	0.207097
17		Batch2	0.535469	0.072328	0.069436	0.218673
18		Batch3	0.479003	0.077375	0.039500	0.173379
19		Batch4	0.581102	0.030989	0.081716	0.264297
20		Batch5	0.449007	0.041666	0.034791	0.193003
21	Epoch5	Batch1	0.358071	0.012281	0.007726	0.284051
22		Batch2	0.384489	0.048140	0.093535	0.191841
23		Batch3	0.390661	0.038681	0.020918	0.232492
24		Batch4	0.424738	0.018279	0.018712	0.107939
25		Batch5	0.355652	0.032140	0.046137	0.262498
26	Epoch6	Batch1	0.327019	0.026536	0.056779	0.277442
27		Batch2	0.411010	0.021211	0.014622	0.228826
28		Batch3	0.454030	0.008986	0.006331	0.296529
29		Batch4	0.454882	0.036895	0.017617	0.228233
30		Batch5	0.371328	0.037801	0.065295	0.291198
31	Epoch7	Batch1	0.437205	0.032438	0.018010	0.165724
32		Batch2	0.356190	0.027671	0.006267	0.222398
33		Batch3	0.350327	0.023559	0.015799	0.206092
34		Batch4	0.323758	0.031195	0.006477	0.215378
35		Batch5	0.468034	0.068337	0.008844	0.163495
36	Epoch8	Batch1	0.459280	0.014680	0.053380	0.209495
37		Batch2	0.373781	0.002587	0.005783	0.256620
38		Batch3	0.305165	0.027237	0.023110	0.110588
39		Batch4	0.486001	0.024209	0.020935	0.140438
40		Batch5	0.254745	0.009955	0.030645	0.147727
41	Epoch9	Batch1	0.310326	0.014629	0.008775	0.234523
42		Batch2	0.291564	0.005057	0.063808	0.157785



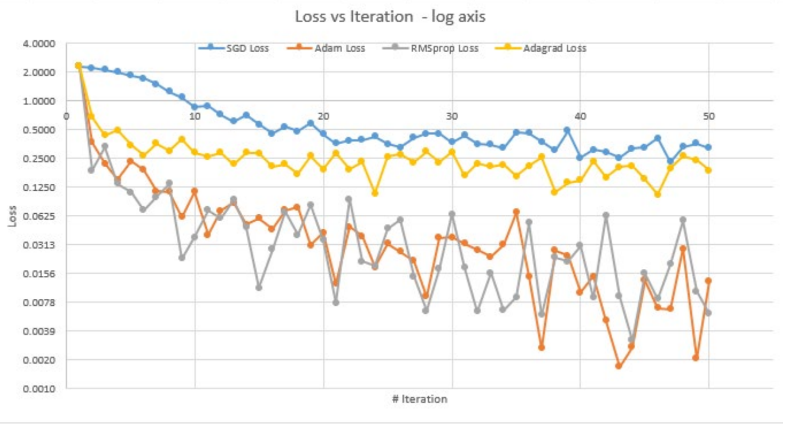


Fig 4. Experiment Result of different optimizers. Data collected via # of iterations.

APPENDIX

The code used in this project is attached in this Appendix. Please note that the part of the code is from Hw-04.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
from torch.autograd import Variable
import time
import sys



class LeNet(nn.Module):
  def __init__(self):
    super(LeNet, self).__init__()
    self.conv1 = nn.Sequential(
      nn.Conv2d(1, 6, 5, 1, 2),
      nn.ReLU(),
      nn.AvgPool2d(kernel_size=2, stride=2),
    )
    self.conv2 = nn.Sequential(
      nn.Conv2d(6,16,5),
      nn.ReLU(),
      nn.AvgPool2d(kernel_size=2, stride=2),
    )

    self.fc1 = nn.Sequential(
      nn.Linear(16*5*5, 120),
      nn.ReLU()
    )
    self.fc2 = nn.Sequential(
      nn.Linear(120,84),
      nn.ReLU()
    )

    self.fc3 = nn.Linear(84, 10)

  def forward(self, x):
    x = self.conv1(x)
    x=self.conv2(x)
    x = x.view(x.size()[0], -1)
    x = self.fc1(x)
    x = self.fc2(x)
    x=self.fc3(x)
    return x  # F.softmax(x, dim=1)


def train(epoch,optimizer):
  model.train()
  for batch_idx, (data, target) in enumerate(train_loader):
    data, target = Variable(data), Variable(target)
    optimizer.zero_grad()
    output = model(data)
    loss = F.cross_entropy(output, target)
    loss.backward()
    optimizer.step()
    if batch_idx % 100 == 0:
      print('Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
        epoch, batch_idx * len(data), len(train_loader.dataset),
           100. * batch_idx / len(train_loader), loss.item()))


def test():
  model.eval()
  test_loss = 0
  correct = 0
  for data, target in test_loader:
    data, target = Variable(data), Variable(target)

    output = model(data)
    test_loss += F.cross_entropy(output, target,
size_average=False).item()
    pred = output.data.max(1, keepdim=True)[1]
    correct += pred.eq(target.data.view_as(pred)).cpu().sum()

  test_loss /= len(test_loader.dataset)
  print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{}
({:.0f}%)\n'.format(
    test_loss, correct, len(test_loader.dataset),
    100. * correct / len(test_loader.dataset)))
```

```
if __name__ == '__main__':


    epochs = 10
    batch_size = 128
    lr=0.001

    train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('./minst', train=True, download=True,
                       transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,))
                       ])),
        batch_size=batch_size, shuffle=True)

    test_loader = torch.utils.data.DataLoader(
        datasets.MNIST('./minst', train=False,
transform=transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.1307,), (0.3081,))
        ])),
        batch_size=batch_size, shuffle=True)



    sgd = torch.optim.SGD
    adam=torch.optim.Adam
    rmsprop=torch.optim.RMSprop
    adagrad=torch.optim.Adagrad
    sys.stdout  = open("output3.txt","w")

    for name,optimizer in
zip(['SGD','Adam','RMSprop','Adagrad'],[sgd,adam,rmsprop,adag
rad]):
        model = LeNet()
        optimizer=optimizer(model.parameters(),lr=lr)
        print('#'*40,'\n','Optimizer:\n',name,'\n','#'*40,)
        start = time.time()
        for epoch in range(1, epochs + 1):
            train(epoch,optimizer)
            test()
            end = time.time()
            delta = end - start
            print('Total runtime: ',delta,'s')
    sys.stdout.close()
```

Thanks for reading.