COMPARISON OF FIRST ORDER OPTIMIZATION METHODS FOR DEEP LEARNING

Aditya Chindhade

DECEMBER 9, 2017

CARNEGIE MELLON UNIVERSITY
5000 Forbes Ave, Pittsburgh, PA 15213

Contents

1. Abstract:

The project described below gives a comparative study of the various first order optimization methods used in minimizing the loss function in a convolutional neural network (CNN) implemented on the MNIST dataset. The mini-batch length is varied and the loss, accuracy and neural network training times for each optimization method are studied.

2. Introduction

A convolutional neural network (CNN) is a type of an artificial neural network (ANN) which is specifically suited to classify image datasets due to its interconnected structure. The layer-wise convolution operation is similar to the way neurons respond to visual stimuli. Each of the layers involve the minimization of a loss function with respect to the weighing parameters which determines the accuracy of the fitting model.

The optimization step described above plays a significant role in the overall accuracy of the model and hence is of interest to machine learning engineers. Here, the overall accuracy and loss minimization of the model has been studied for different first order optimization methods available in standard machine learning libraries such as Keras™ and TensorFlow™. These involve computation of only the gradient (Jacobian) of the cost function and hence are known as first order methods.

Methods which account for second derivative information (Hessian/hessian approximation) are called second order methods. In order to account for curvature information about the function, second order methods such as BFGS and SR-1 have been proposed. This second derivative information allows the minima to be reached in much fewer steps than the first order methods. However, each step of a second order method is computationally expensive and time consuming. In addition to this, due to the large size of the dataset (~60,000 for MNIST), Hessian inverse calculations being tedious, are avoided and replaced by Hessian approximations.

An interesting tradeoff can be identified here: first order methods require more steps to convergence but involve faster steps, whereas second order methods involve fewer, more time-consuming steps. As a result, the choice of an appropriate optimization method is crucial.

3. First order methods:

a. Stochastic gradient descent (SGD):

SGD is one of the most popular optimization techniques in machine learning circles[1]. It works by computing the gradients over a batches or subsets of the dataset known as mini-batches. The size of these mini batches can affect the accuracy of the overall method. This approach is based on the analytical steepest descent technique used in optimization.

The update formula for SGD is as follows:

$$w^{k+1} = w^k - \alpha \nabla f(w^k), where\ \alpha, the\ step\ size\ is\ also\ known\ as\ the\ learning\ rate$$

b. Stochastic gradient descent with momentum:

SGD with momentum involves a linear combination of the previous update on the weight and the gradient while calculating the update on w.

The update formula for SGD with momentum is given by Rumelhart et.al.[2] is as follows:

$$w^{k+1} = w^k - \alpha \nabla f(w^k) + \beta \Delta w, where\ \alpha\ the\ learning\ rate$$

c. Adam:

Adam stands for Adaptive moment estimation and involves the weighted averages of the gradients and second moments of the gradients. The algorithm for Adam was given by Kingma et. al.[3] as follows:

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Adadelta:

Adadelta was first reported in 2012 at Google® by Zeiler et.al.[4] and the algorithm is as follows:

Algorithm 1 Computing ADADELTA update at time t

Require: Decay rate ρ, Constant ϵ
Require: Initial parameter x_1
1: Initialize accumulation variables $E[g^2]_0 = 0$, $E[\Delta x^2]_0 = 0$
2: **for** $t = 1 : T$ **do** %% Loop over # of updates
3: Compute Gradient: g_t
4: Accumulate Gradient: $E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho)g_t^2$
5: Compute Update: $\Delta x_t = -\frac{\text{RMS}[\Delta x]_{t-1}}{\text{RMS}[g]_t}\, g_t$
6: Accumulate Updates: $E[\Delta x^2]_t = \rho E[\Delta x^2]_{t-1} + (1-\rho)\Delta x_t^2$
7: Apply Update: $x_{t+1} = x_t + \Delta x_t$
8: **end for**

4. Experimental section:

A convolutional neural network was implemented using the standard Keras™ library and the experiments performed were matched with the results of Ramamurthy et al.[5] The layer-structure of the network is as follows:

a. Convolutional layer
b. Batch normalization layer
c. Max Pooling layer
d. Convolutional layer
e. Batch normalization layer
f. Max pooling layer
g. Batch normalization layer

The activation function used was ReLU, loss function used was categorical crossentropy.

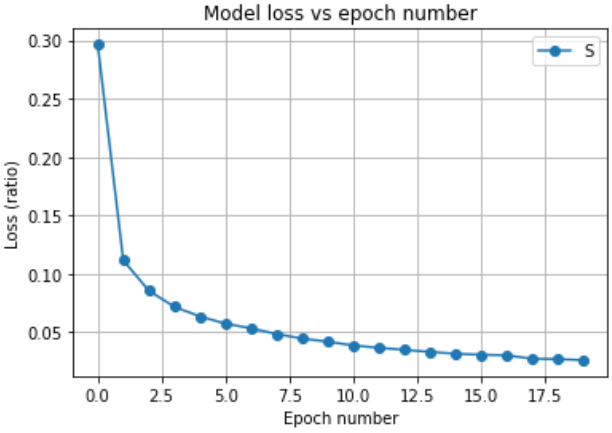
The network was trained over 20 epochs for the MNIST training dataset with mini-batch sizes of 20 & 32.

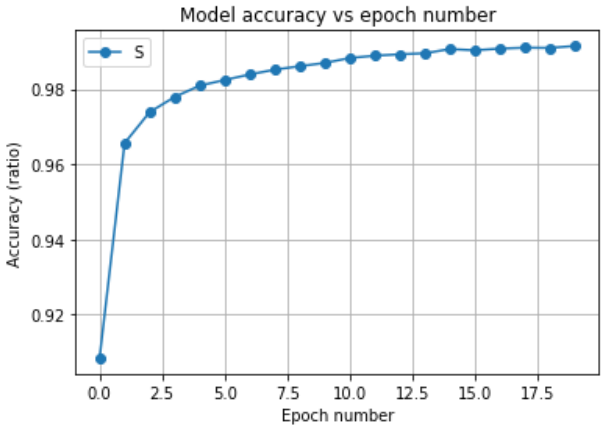
The loss and accuracy was recorded for each epoch and the training time for the network was noted.

The network was tested on the MNIST test dataset and the loss, accuracy and testing times were recorded.

5. Results: Mini-batch size = 20

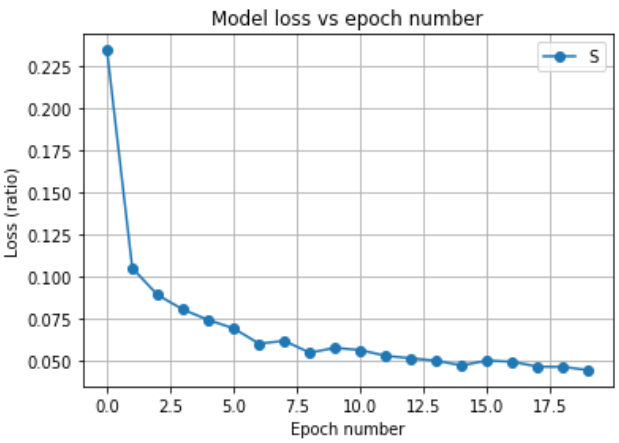
a. Stochastic Gradient Descent (SGD):

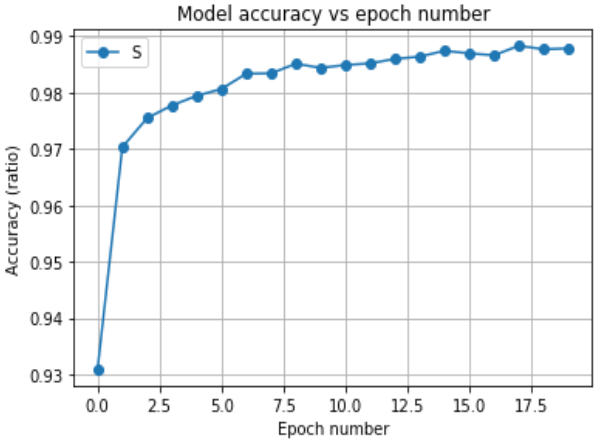




- Neural network training time is: 4434.5606 seconds
- When evaluated on the MNIST test dataset, the loss is: 0.0210
- When evaluated on the MNIST test dataset, the accuracy is: 0.9931
- Evaluation and plotting runtime is: 15.8835 seconds

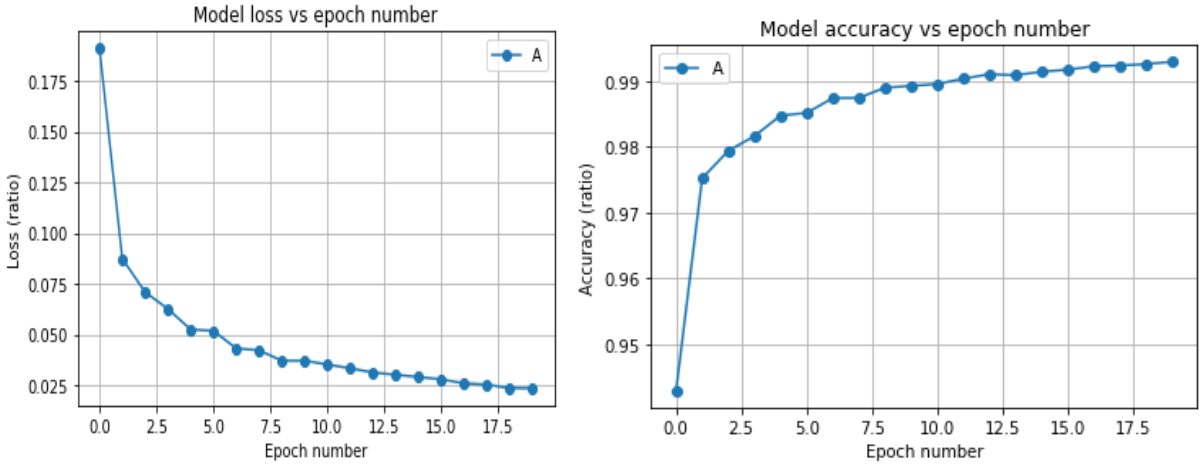
b. Stochastic Gradient Descent with momentum:





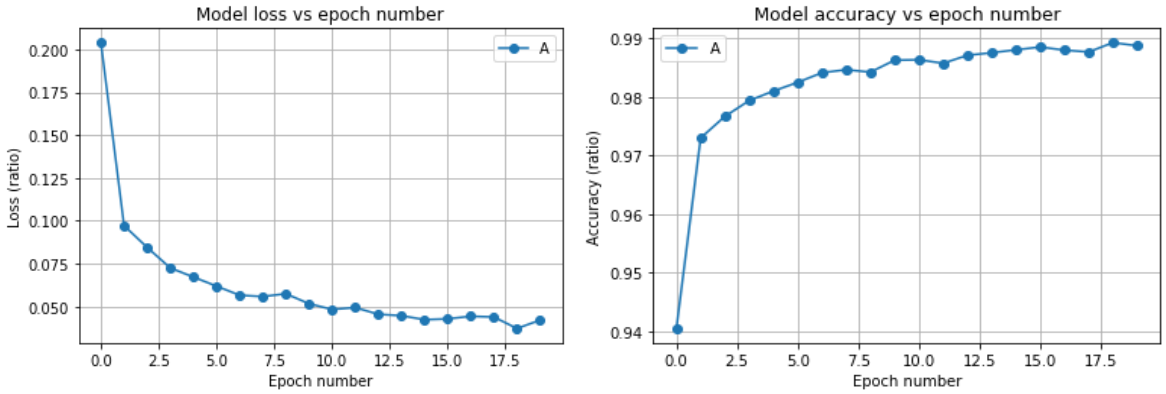
- Neural network training time is: 4441.6443 seconds
- When evaluated on the MNIST test dataset, the loss is: 0.0391
- When evaluated on the MNIST test dataset, the accuracy is: 0.9919
- Evaluation and plotting runtime is: 15.6856 seconds

c. Adam optimizer:



- Neural network training time is: 4554.9762 seconds
- When evaluated on the MNIST test dataset, the loss is: 0.0281
- When evaluated on the MNIST test dataset, the accuracy is: 0.9936
- Evaluation and plotting runtime is: 15.9780 seconds

d. Adadelta:



- Neural network training time is: 2466.4945 seconds
- When evaluated on the MNIST test dataset, the loss is: 0.0331
- When evaluated on the MNIST test dataset, the accuracy is: 0.9930
- Evaluation and plotting runtime is: 8.2851 seconds

e. Overall comparison of first order methods with literature:

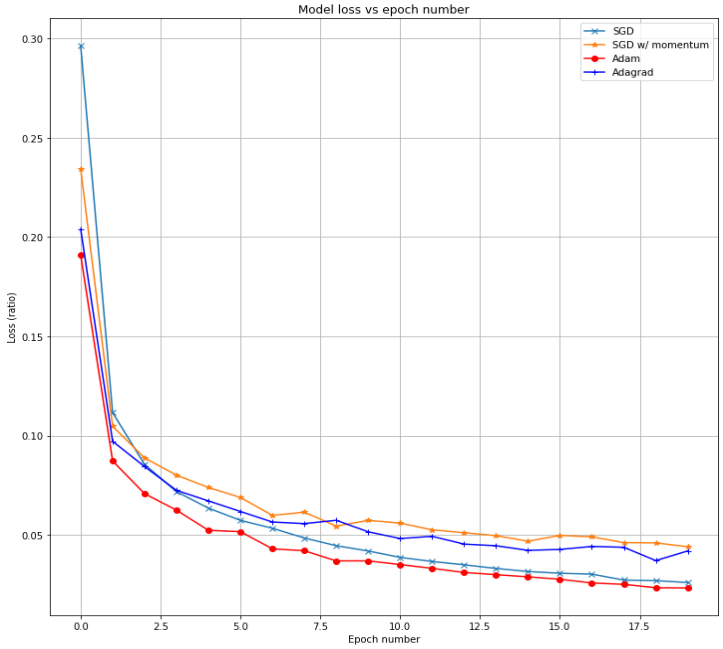


Figure 1 Loss vs Epoch on implemented network

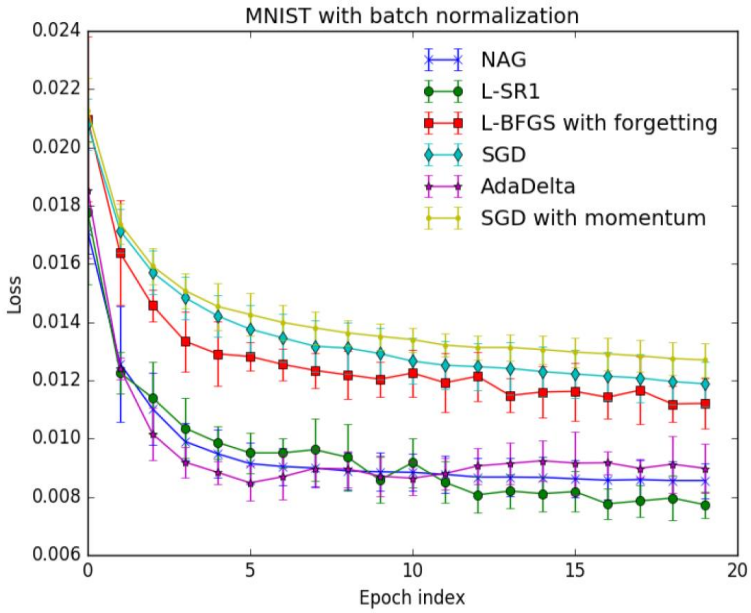
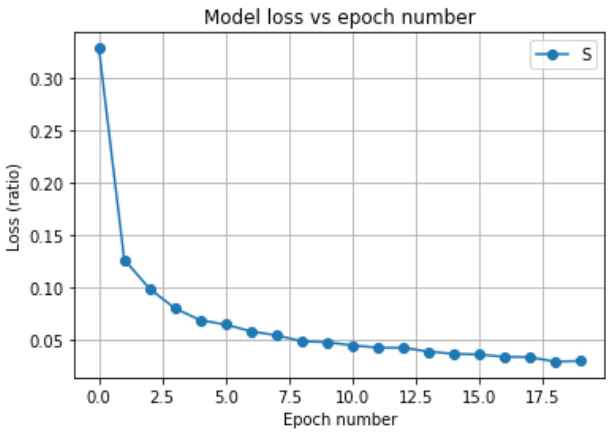
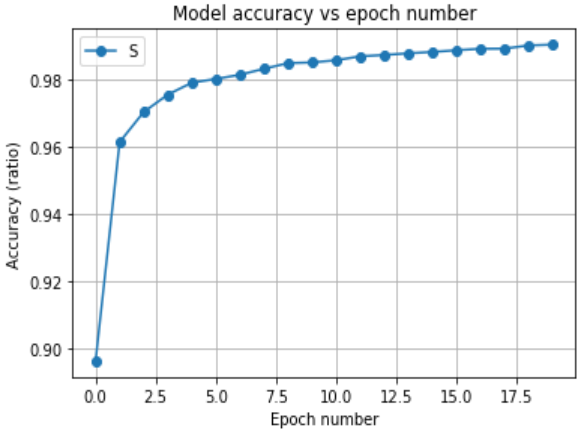


Figure 2 Loss vs epochs as reported by Ramamurthy et.al

6. Mini-Batch size = 32

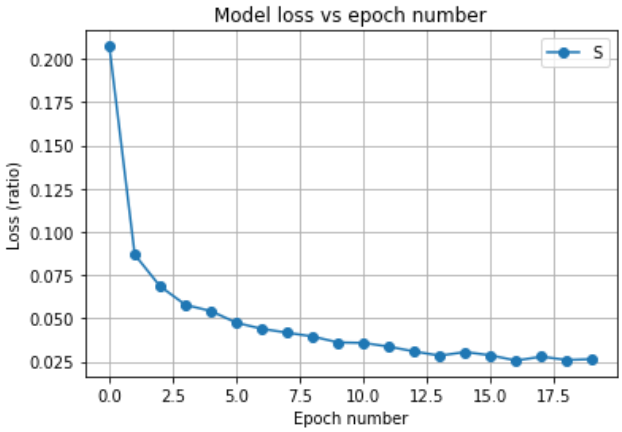
a. Stochastic gradient descent (SGD):

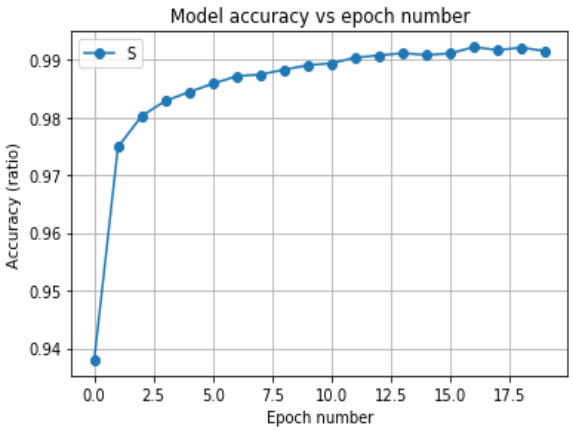




- Neural network training time is: 2412.8643 seconds
- When evaluated on the MNIST test dataset, the loss is: 0.0218
- When evaluated on the MNIST test dataset, the accuracy is: 0.9929
- Evaluation and plotting runtime is: 8.2898 seconds

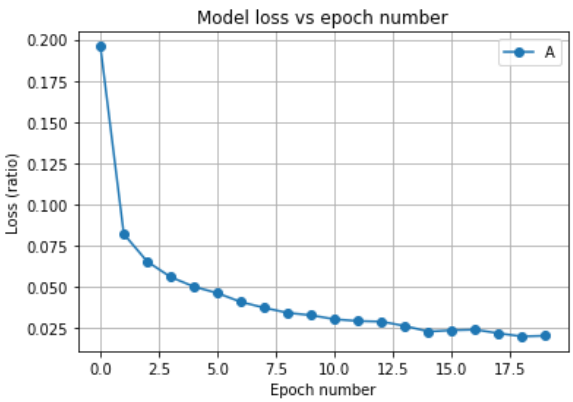
b. Stochastic gradient descent with momentum:

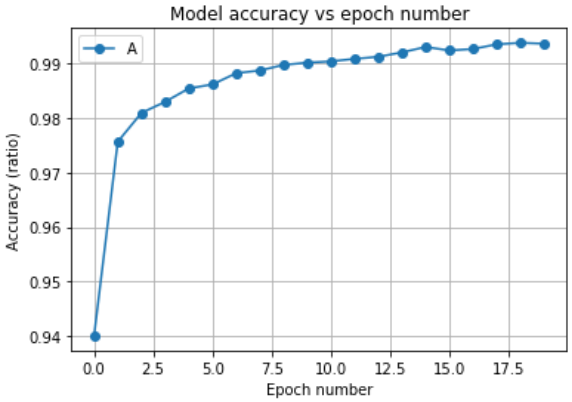




- Neural network training time is: 2238.8152 seconds
- When evaluated on the MNIST test dataset, the loss is: 0.0323
- When evaluated on the MNIST test dataset, the accuracy is: 0.9923
- Evaluation and plotting runtime is: 8.1990 seconds

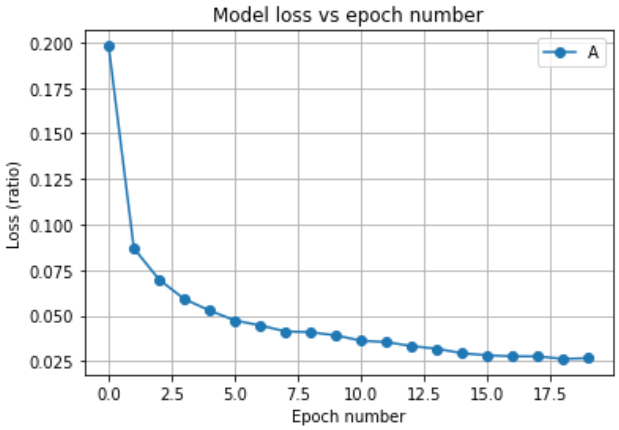
c. Adam:

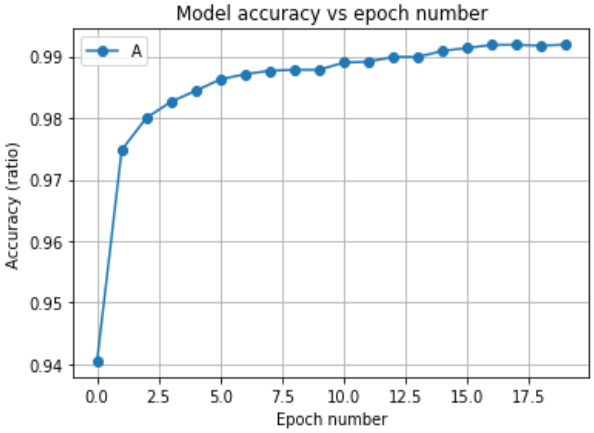




- Neural network training time is: 2327.2067 seconds
- When evaluated on the MNIST test dataset, the loss is: 0.0244
- When evaluated on the MNIST test dataset, the accuracy is: 0.9932
- Evaluation and plotting runtime is: 8.6509 seconds

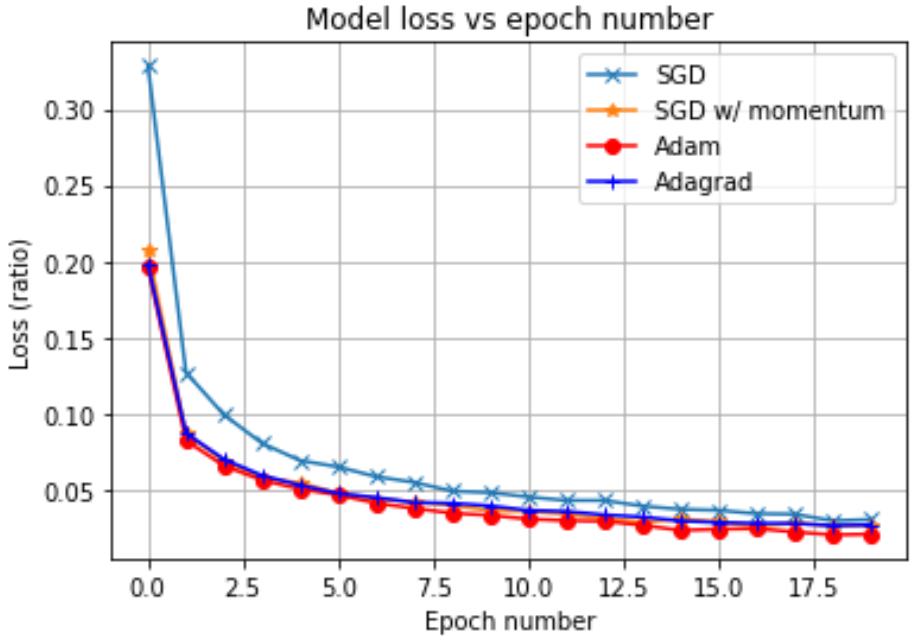
d. Adadelta:

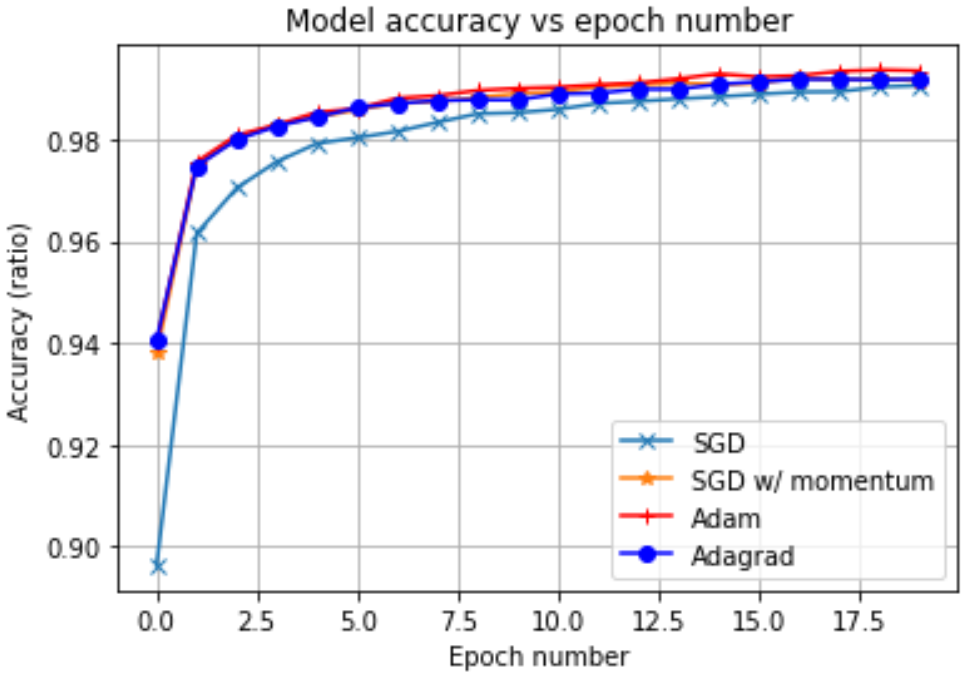




a. Neural network training time is: 2175.5490 seconds
b. When evaluated on the MNIST test dataset, the loss is: 0.0243
c. When evaluated on the MNIST test dataset, the accuracy is: 0.9928
d. Evaluation and plotting runtime is: 8.2756 seconds

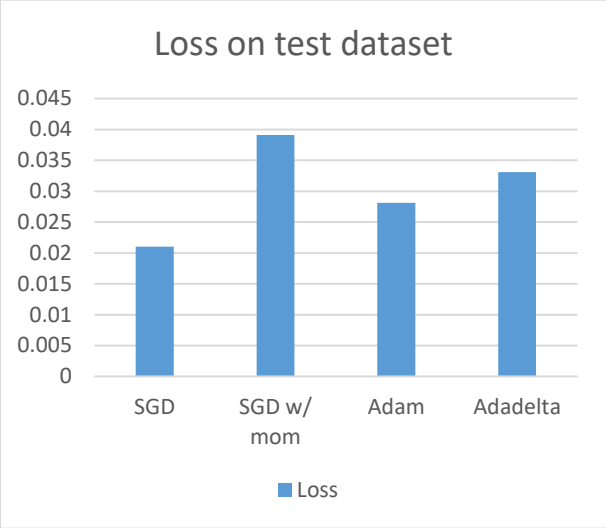
e. Overall comparison of first order methods:

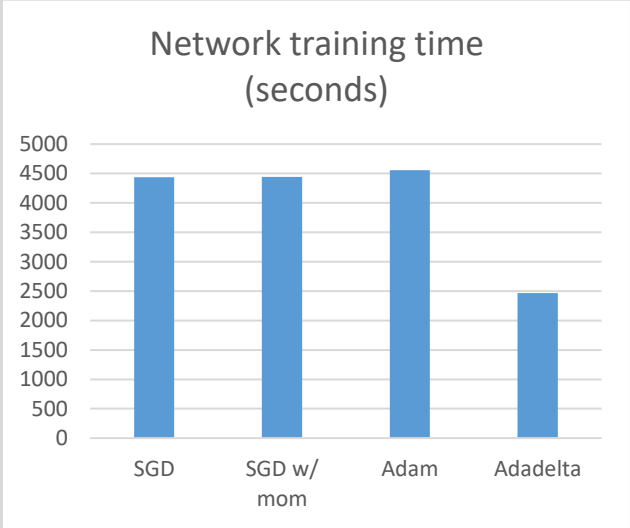




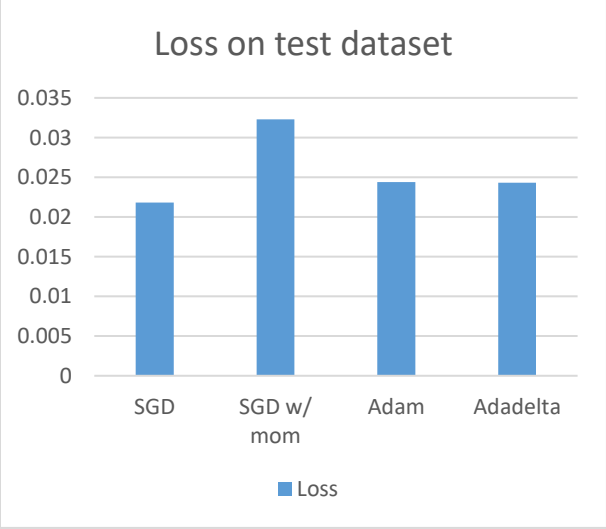
f. Results from test datasets and network training time:

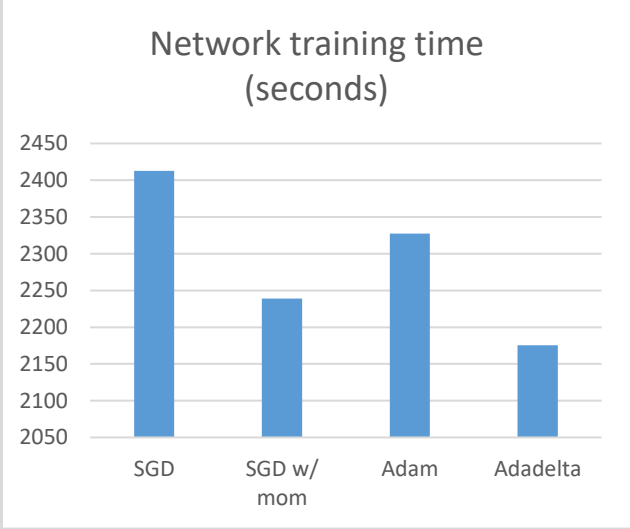
g. Batch size = 20





h. Batch size = 32





i. Discussion:

The plots from both the batch sizes (20, 32) show a consistent decrease in the loss with an increase in the number of epochs except for a few outlier points. This indicates the overall updates in the model weights are leading to convergence. The graph of loss plotted against number of epochs is in close agreement to the one reported by Ramamurthy et. al. for the four first order methods.

For experiments on the test datasets, the stochastic gradient descent method consistently shows the lowest loss value among the four optimizers for both batch sizes. However, the experiment was performed only once due to long runtime (~4 hrs) and additional experiments with error bars would give us a clearer picture.

The network training time drops significantly (~1/2) when the batch size is increased from 20 to 32. In both the cases, the adadelta method takes the least amount of time. In case of batch size of 32, SGD with momentum is the second fastest network to be trained. In case of the former case, the significant drop in the training time corresponding to adadelta is an experimental outlier due to the fact that the code segment was run without any parallel processes running on the computer.

All experiments were performed on a Lenovo Thinkpad T470p machine with an Intel™ i7 processor at 2.9 GHz and 16GB of memory and 64 bit architecture without active GPUs.

j. Conclusion:

Experiments indicate that among first order methods, SGD provides the lowest loss value for the given MNIST dataset test dataset. We can also conclude that the network training time decreases significantly as the number of batch sizes. This happens because the variance among the updates from the larger batch is lesser as compared to the large variance in average gradient updates from the smaller batch. As a result, the step size can be made larger, hence lowering the time to reach the minima.

k. Future work:

Second order optimization methods such as BFGS and SR-1 are currently unavailable within standard machine learning libraries such as Keras™ and TensorFlow™. Extensive effort was made by me to incorporate the BFGS and SR-1 updates into the libraries, however, it involved passing multiple arguments into a wrapper function around the algorithms and many of these were to be extracted by accessing the Keras and TensorFlow backend. A detailed

7. References:

1. Bottou, L. Large-Scale Machine Learning with Stochastic Gradient Descent. *Proc. COMPSTAT'2010* 177–186 (2010). doi:10.1007/978-3-7908-2604-3_16
2. Rumelhart, D. E., Hinton, G. E. & Williams, R. J. Learning representations by back-propagating errors. *Nature* **323,** 533–536 (1986).
3. Kingma, D. P. & Ba, J. Adam: A Method for Stochastic Optimization. 1–15 (2014). doi:http://doi.acm.org.ezproxy.lib.ucf.edu/10.1145/1830483.1830503
4. Zeiler, M. D. ADADELTA: An Adaptive Learning Rate Method. (2012).
5. Ramamurthy, V. & Duffy, N. L-Sr1: a Second Order Optimization Method for Deep Learning. *Iclr* 1–15 (2017).

8. Appendix:

MNIST is a database of handwritten digits publicly available at http://yann.lecun.com/exdb/mnist/