**Applying Stochastic Limited Memory Nesterov’s Accelerated Quasi-Newton Method to a Neural Network**

Kyle Fletcher

**Introduction and Background**

Neural Networks are currently a very large area of research since they are applicable to problems across an innumerable number of different domains. Deep networks specifically can be considered as universal approximators, with the ability to learn any decision boundary among a set of data points. Learning a neural network involves minimizing an objective function known as a loss function. Even if a neural network has multiple outputs, they are combined into a single scalar output value by the objective loss function. These loss functions are often highly non-linear and non-convex, making finding a minimum to be tricky.

A typical neural network involves a very large number of weights which are seen as the primary inputs to the loss function along with the training data, giving the function a high dimensionality. The loss function ideally includes all the training data so that its minimization is done with consideration of the whole sample space. Optimization methods using this form of loss function are known as batch methods. Optimization methods in neural networks involve passing training data through the network and backward passing it to find the gradients of the weights in the network. Often there are at least thousands or more training samples for a network. With a large network and a lot of training data, batch methods become too expensive. To solve this, the loss function is instead calculated with a minibatch of training data, which changes on each step of the minimization, and looks like

where is the set of weights, is the set of training samples, is the mini-batch and is the mini-batch size [1]. Optimization methods using this form are known as stochastic methods and are what most neural networks use due to high network and sample size.

The actual optimization methods used for neural networks are most commonly first order methods, meaning they only use the information from the gradient of the loss function with respect to the weights of the network. The simplest form of this is stochastic gradient descent, where the weights are updated by the negative gradient at a particular learning rate. Gradient descent is often way too slow at converging upon an actual minimum, so there have been many other first order methods developed. Among these are momentum methods which include a scaled down velocity vector of the previous step taken in determining the minimization’s next step. Where gradient descent would slow down in directions of low curvature, momentum increases the rate of reduction when the direction is consistent[2]. The optimization method we’re using is partially based off a similar method to momentum, Nesterov’s Accelerated Gradient, NAG. The difference between the two is NAG calculates the gradient after applying the velocity vector from momentum to the current position. This makes NAG more stable and responsive to changes in the objective function then classical momentum [2].

Another way to improve convergence over gradient descent is to use second order methods. Second order methods use curvature information, like the hessian to improve the step direction. The Newton method is the most basic second order method and involves multiplying the gradient with the negative inverse of the hessian to determine the step direction for minimization. Given the right conditions this method achieves quadratic convergence. The problem with this is that in large problems the calculation of the hessian and its inverse is too expensive. The solution to this is to use a quasi-newton method. Quasi-newton methods approximate the hessian using only information from the gradient of the objective function. These methods can produce superlinear convergence under the right conditions, which is not as good as Newton but better than gradient descent. The BFGS method is the most popular quasi-newton method, which attempts to directly approximate the inverse of the hessian, which is what is used in calculating the step direction [3]. The optimization method we’re using is also partially based off BFGS.

Both the BFGS and the NAG methods have been combined to form Nesterov’s Accelerated Quasi-Newton method, NAQ. This method attempts to introduce the NAG updates into the BFGS method to combine the advantages of momentum with the advantages of using curvature information [4]. This method is however originally utilized for full batch updates. In the case of large neural networks, full batch updates are too expensive. To apply NAQ to a large network, it would need to be done stochastically. However, since it is based off BFGS it cannot be used stochastically without modification due to invalidity of a line search and added noise into the hessian approximation in a stochastic setting. To solve this problem, a stochastic “online” BFGS, oBFGS, was proposed for convex optimization with modifications to address these problems [5]. These modifications have been combined with NAQ makes a stochastic, “online” version of NAQ possible, oNAQ [1].

There’s one more potential issue with applying this method to deep learning problems. In large neural networks there can be millions of individual weight parameters to be optimized. In this case, if the hessian approximation from a quasi-newton method becomes dense over time, it may take up too much memory. This can be avoided by calculating the hessian approximation using only curvature information from only a set of the latest iterations. This concept can be applied to BFGS resulting in a limited memory BFGS, LBFGS [3]. For oNAQ, it was also proposed that the BFGS portion of the method be replaced with LBFGS for networks with a high number of parameters, resulting in oLNAQ, a stochastic, limited memory version of NAQ [1]. This is the method we’re implementing and testing.

**oLNAQ Algorithm and Properties**

This algorithm is specifically based off the stochastic limited memory BFGS method combined with Nesterov’s Accelerated Gradient. Traditional BFGS approximates the inverse hessian at each step using an initial approximation , and using the update

where and . For the limited memory version of BFGS, LBFGS, this equation is replaced by a 2-loop recursion algorithm using only the most recently calculated and pairs which directly calculates the search direction [3]. The online stochastic version of LBFGS, oLBFGS makes a few changes to the algorithm. First, is calculated using only gradients calculated at the same batch inputs and a term is added. Second, the direction vector is scaled by dividing it by its L2 norm . Third, the traditional line search for step size is replaced by a decay schedule applied to an initial [1]. Finally, for the case of limited memory BFGS, steps 7 and 9 in the 2-Loop recursion algorithm (see algorithm 2) are replaced so that the direction being calculated is scaled incredibly low if it’s the first update, and more robust to sample noise in all other updates [5].

Nesterov’s Accelerated Gradient is the first order algorithm combined with oLBFGS. For this algorithm, update vector is calculated using the momentum term of the previous update [4].

In order to apply these changes with the momentum term to oLBFGS, the and are replaced with terms and respectively [1].

To get oLNAQ we simply use and in place of and in oLBFGS and the 2-loop recursion algorithm, along with adding the constant . This results in the full oLNAQ algorithm with limited memory updates below [1], [5].

**Algorithm 1:** oLNAQ

**Initialize:** minibatch

1. while do
2. Calculate using Algorithm 2, the 2-loop recursion
3. Keep last pairs of and
4. end while

**Algorithm 2:** 2-Loop Recursion Direction Update for oLNAQ

**Require:**  memory size , curvature pairs

1. for do
2. end for
3. if then
4. else
5. end if
6. for do
7. end for

As can be seen there are a few constants and hyperparameters that must be chosen for these algorithms. Their selection will be described in the next section.

The original NAQ has been proven to satisfy the secant equation in a similar manner to BFGS. Furthermore, the updates to the inverse hessian approximation are proven to remain symmetric positive definite. Due to these properties it is claimed the original NAQ has similar convergence to other quasi-newton methods [4]. While the stochastic form of NAQ may adversely affect some of these properties, the simulation results in the original paper show it to still be an effective optimizer [1].

The computational and storage costs of traditional BFGS are and respectively, where is training size, is parameter size, and is the variable number of iterations for a successful line search. These costs in traditional NAQ are and respectively. Traditional NAQ has a higher cost since an extra gradient with the momentum term must be calculated. However, the computation and storage costs of oLBFGS and oLNAQ are the same, and respectively. The computation advantage of BFGS is eliminated due to the necessity for 2 unique gradient calculations in its stochastic form [1].

**Dataset, Network, and Implementation Details**

In the original paper on oLNAQ, different sizes of the classic MNIST dataset are used to test this algorithm’s capabilities on traditional neural network classification tasks. MNIST is a dataset of grayscale images of handwritten digits classified as 0 through 9. In the original work, oLNAQ was shown to perform well on both a version of MNIST with an 8x8 set of inputs, and a version with a 28x28 set of inputs. The algorithm was tested as an optimizer on convolutional neural networks for these datasets [1].

For this implementation of oLNAQ the dataset used was the Fashion-MNIST. This dataset consists of 28x28 grayscale images of various clothing items. Similar to MNIST, there are 10 classes of data, each representing a type of clothing item. The dataset is already divided into a training set of 60,000 images and a testing set of 10,000 images. Fashion-MNIST is claimed to be more difficult to classify than the original MNIST dataset [6]. This dataset was chosen because it’s similar in structure to the dataset used in the original oLNAQ paper. However, being more difficult to classify, it may require a larger, more complex network to perform well. Usually with image classification convolutional neural networks must be used to find features in an image in a location independent way. However, fully linear neural networks have been shown to achieve high accuracy on the original MNIST, so in some ways it can be considered too easy of a dataset to rely on [6]. Using a dataset such as Fashion-MNIST takes the algorithm one step closer to being used in real world applications beyond simple benchmarks.

The neural network architecture used in this implementation was still a relatively simple convolutional architecture. There were 2 convolutional layers, each with 3x3 kernels. The first convolutional layer had 16 output filters and the second had 32 output filters. No padding was done so these convolutional layers transformed the input size to 24x24x32. Then a 2d max pool of size 2x2 and stride 2 was applied, dropping the dimensions to 12x12x32. Next, dropout was performed at a probability of 0.25 to lessen the effect of overfitting. At this point the data was flattened and passed through a fully connected layer with 100 neurons, followed by an output layer of 10 neurons. The result of all these layers was a model with 466710 trainable parameters. Every layer besides the output layer used ReLU activation, and the output layer used softmax. This architecture mainly differed from the ones used in the original paper by a larger number of output filters at the convolutional layers and an additional fully connected layer [1].

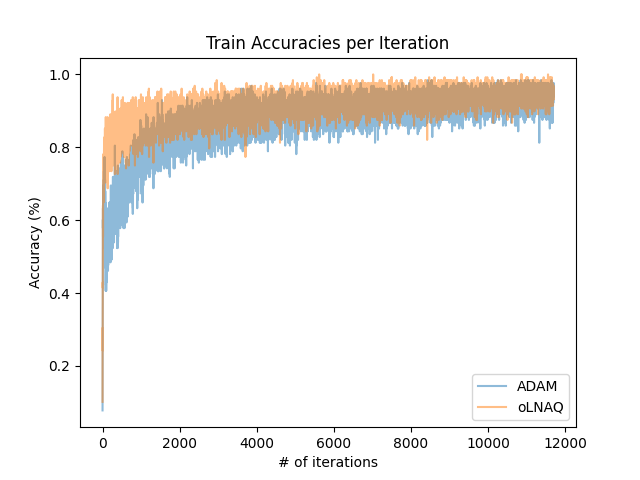
There were several hyperparameters and algorithm parameters that needed to be chosen for the network. The network was trained over 25 epochs since at that amount, clear trends can be seen but execution doesn’t take too long. Batch size was chosen to be 128 since that size was shown to work the best for oLNAQ in the original paper results. The momentum parameter most used for the classification networks was so that was chosen as the momentum parameter. Memory size for the limited memory algorithm was chosen to be the same as in the original at It is also mentioned that for the decaying learning rate, the initial step size is often chosen to be 1.0, so was used for initial learning rate [1]. In the case of oBFGS the epsilon used for scaling the first direction calculation is set to , so it is set to be the same here. Additionally, during experiments with Conditional Random Fields, which have over parameters, the model trust region parameter was set to be 1.0 [5]. Given that our neural network models also have a very high order of parameters, was chosen to be 1.0.

**Results**

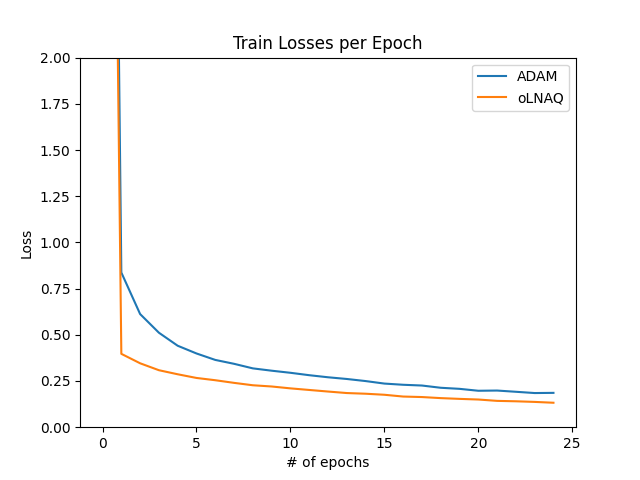
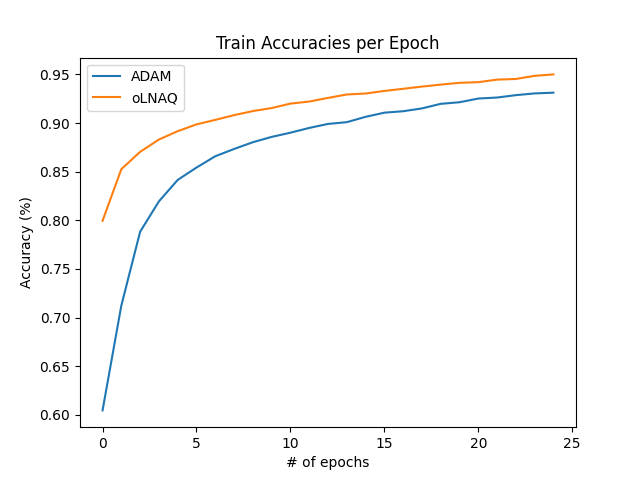
The results are all based on values recorded during training. 2 separate models were trained, one on Adam and one on oLNAQ. Adam is a first order optimization algorithm that utilizes momentum adjusted by two separate decay rates [7]. The goal was to compare it to oLNAQ considering its one of the most popular optimizers for neural network training.

After each iteration, the cross-entropy loss and accuracy were recorded. These values were also recorded at the end of each epoch. The test set of data provided for the Fashion-MNIST was used as validation data, and at the end of each epoch the accuracy on the whole validation set was calculated and recorded. The results from both models are shown below in figures 1-3.

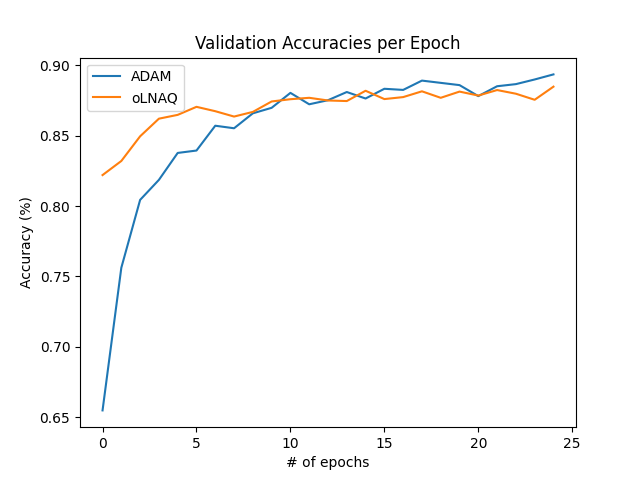
A graph of a graph showing a number of losses

Description automatically generated with medium confidence

*Figure 1:* Training Results per Iteration  
*Left:* Cross-entropy Loss per Iteration  
*Right:* Accuracy per Iteration



*Figure 2:* Training Results per Epoch  
*Left:* Cross-entropy Loss per Epoch  
*Right:* Accuracy per Epoch



*Figure 3:* Validation Accuracies per Epoch

Figures 1-3 show some clear trends with regards to oLNAQ. Its training accuracy manages to reach around 95% while its validation accuracy hovers just below 90%. It appears to drop the loss very quickly, but convergence slows down a lot over time. From these charts we see that it outperforms Adam consistently in training loss, though the benefit is mainly seen in early epochs and Adam begins to catch up towards the end of training. This may be partially due to the use of a set decay schedule for the learning rate in oLNAQ. We see from the validation accuracies that oLNAQ has a problem of overfitting as the epoch count increases, and Adam eventually catches up and outperforms it here. The clearest observation here is that oLNAQ initially gets close to the solution much faster than a first order method such as Adam, but it does not outperform Adam to the same degree in later epochs.

**Conclusion/What I Learned**

The results show that it is possible to use a quasi-newton method such as oLNAQ for training a neural network. There are situations in the original paper and here where this method is more successful than the commonly used optimizers for neural networks. The process of researching this material has shown me that there are a lot of differences between designing an optimizer for something like a neural network vs minimizing a convex objective function with full batches and much less parameters. Through the complexity of the oLNAQ algorithm we see that there’s a lot of things that must be considered in a stochastic or limited memory setting with a non-convex function. I did notice when building the models that the optimizer for oLNAQ ran much slower. That is likely due partly to an inefficient implementation but also due to the necessary extra calculations for the algorithm, which is why the original paper plots values over time in many cases in addition to iterations. These additional calculations must be considered when evaluating the success of an algorithm like oLNAQ. Also, through the implementation it was interesting to see how modifying the hyperparameters could seriously affect the efficiency and performance of the model. My impression from these tests is that it highly depends on the type and size model being optimized and how quickly training convergence is desired as to whether a quasi-newton method like oLNAQ would be useful in a deep neural network context.

**References**

[1] S. Indrapriyadarsini, S. Mahboubi, H. Ninomiya, and H. Asai, “A Stochastic Quasi-Newton Method with Nesterov’s Accelerated Gradient,” in *Machine Learning and Knowledge Discovery in Databases*, U. Brefeld, E. Fromont, A. Hotho, A. Knobbe, M. Maathuis, and C. Robardet, Eds., Cham: Springer International Publishing, 2020, pp. 743–760. doi: 10.1007/978-3-030-46150-8\_43.

[2] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, “On the importance of initialization and momentum in deep learning,” in *Proceedings of the 30th International Conference on Machine Learning*, PMLR, May 2013, pp. 1139–1147. Accessed: Apr. 20, 2024. [Online]. Available: https://proceedings.mlr.press/v28/sutskever13.html

[3] *Numerical Optimization*. in Springer Series in Operations Research and Financial Engineering. Springer New York, 2006. doi: 10.1007/978-0-387-40065-5.

[4] H. Ninomiya, “Neural network training based on quasi-Newton method using Nesterov’s accelerated gradient,” in *2016 IEEE Region 10 Conference (TENCON)*, Nov. 2016, pp. 51–54. doi: 10.1109/TENCON.2016.7847957.

[5] N. N. Schraudolph, J. Yu, and S. Günter, “A Stochastic Quasi-Newton Method for Online Convex Optimization,” in *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics*, PMLR, Mar. 2007, pp. 436–443. Accessed: Apr. 20, 2024. [Online]. Available: https://proceedings.mlr.press/v2/schraudolph07a.html

[6] H. Xiao, K. Rasul, and R. Vollgraf, “Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms.”

[7] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” arXiv.org. Accessed: Apr. 24, 2024. [Online]. Available: https://arxiv.org/abs/1412.6980v9