This project is a team effort: Tuan Tran, Alec Diraimondo

Second Order Optimization Methods with Application to Deep Neural Nets

CS595 - Fall 2020

A. Abstract:

Optimization is the backbone of learning in modern Machine Learning methods. In general, there exists two main categories of optimization algorithms: first order and second order methods. First-order stochastic methods have been regarded as the state-of-the-art in large-scale machine learning optimization primarily due to their very efficient per-iteration complexity. This makes them suitable for practical applications, especially in this day and age where data is becoming much larger and widespread. Traditional second order methods used for convex optimization, however, have mostly been overlooked due to the very high computing cost and memory footprint, making it impractical in the context of deep learning.

This project will extend on the previous project about classical second order methods and discuss and experiment with two recent second order optimization methods that have been developed specifically for the problems with optimization in deep learning. These two methods, namely Apollo and AdaHessian, aim to improve on the current state of the art (S.O.T.A) by combining the best of first order methods (feasible time and space complexity) and the best of classical second order methods (beneficial curvature information). In the following sections, we will go over and review the methods and discuss their results.

B. Introduction:

So far, with the ever increasing amount of data, first order methods have emerged as the superior choice in practical settings of deep learning. This is primarily due to their efficiency as well as the need of stochastic sampling of datasets for feasible iterations (datasets are much too large to fit into ram today). This means that at every iteration, one is not computing the gradient but rather an stochastic estimation of the gradient. This estimation of the gradient is subject to noise caused by the variation of each sample or batch. For certain datasets (which each have their own unique probability distribution), this noise and its effect will be varied.

Traditional gradient descent also had specific problems with the structure of neural networks. The biggest issue, namely the vanishing gradient, is caused by the interplay between how gradients of each layer must be computed. Since there is a topological order of the gradients computed at each layer, one must compute the back layers gradient first. These computations are efficiently solved by the backpropagation algorithm, but the consequences are still prevalent in deep networks (ones with many layers).

This is what originally fueled the development of “momentum” into the formulation of the gradient. In essence, if we keep track of m previous iterations (usually m>5) gradients, we can calculate an exponential moving average of the gradient at each iteration. The benefits of this are two-fold: reduce the noisiness of the local gradient from stochastic sampling, whilst improving the vanishing gradient problem and its consequences of underfitting.

To generalize an iterative update to our parameter $\theta \in R^n$, we propose the following equation:

$$\theta_{t+1} = \theta_t - \eta_t \Delta\theta_t \, (1)$$

Where η_t is considered the step size. For gradient descent, or any first-order method: $\Delta\theta_t = g_t$ Where g_t is the gradient (or stochastic gradient) computed at iteration t. Second order methods improve each iterative step by taking into account the second order taylor series expansion term. The matrix of the second-order partial derivatives for any multivariate function is known as the Hessian (denoted as H). In this context then, our parameter update $\Delta\theta$ will be $\Delta\theta = H_t^{-1} g_t$ Thus reformulating (1) with respect to a second-order method:

$$\theta_{t+1} = \theta_t - \eta_t H_t^{-1} g_t \, (2)$$

Since the first order methods only use gradient information to construct the next training iteration, they are often not comparable to second order methods in terms of theoretical convergence asymptotes. In applications to unconstrained convex optimization; gradient descent is said to have linear convergence, whereas the second-order method Newton's Method boasts quadratic convergence.

Additionally, the first order methods may require a series of non-trivial fine-tuning. A lot of efforts have gone into coming up with a more adaptive approach for the first order optimization algorithms to overcome the needs for extensive fine-tuning. As a result, numerous adaptive algorithms were introduced such as Adagrad [1], Adadelta [2] and most notably Adam [3]. These algorithms were shown to produce much better solutions in certain tasks. However, even though they are less sensitive to changes in hyperparameters, through empirical evidence, these methods still require more tuning than ideal.

Compared to first order methods, the second order algorithms have the potential to achieve better results due to adaptation of curvature information. The benefit of this is often two-fold. First, it allows second order methods to not be dependent on the condition number of the problem, whereas the same cannot be said for Gradient methods [4]. Second, it allows adaptively adjusting the step (both direction and magnitude) based on the curvature throughout the training process, leading to potentially even better quality solutions than the adaptive first order algorithms as well as significantly increased robustness to hyperparameter tuning. The main idea underlying second order methods involves **preconditioning** the gradient vector (the step/update) before using it for weight update (as we expressed in (2)). The reason this is better is because for a general problem, different parameter dimensions exhibit different curvature properties. For example, the loss could be very flat in one dimension but sharp in another, and thus we ideally want to take different steps for these dimensions: bigger steps for flatter regions and smaller steps for sharper ones [5]. Unfortunately, classical second order methods like Newton's method are less popular as they are prone to computation and memory issues due to the construction of the Hessian. Moreover, second order methods only really work well if Hessian is positive definite everywhere, meaning the error curve is convex. In the nonconvex landscape of neural networks this need not hold true.

However, due to the advances both in the algorithmic and hardware side, it's becoming more feasible to consider second order algorithms for learning in practical applications, even for non-convex problems. More concretely, there have been various recent attempts at making second order algorithms more efficient and practical. Recently, Ma [6] introduced Apollo and Yao et al. [5] introduced AdaHessian, both being adaptive second order optimizers to address the same shortcomings of computation and memory cost. These second order approaches are referred to as Quasi-Newton methods, as they estimate the Hessian in some way instead of explicitly calculating it. In the general case, we refer to the "curvature matrix" as B. As an example, In newton's method B = H (where H is the Hessian). Any such method where B != H, is referred to as a quasi-newton method. Thus, an even more general form of a second order update can be expressed as:

$$\theta_{t+1} = \theta_t - \eta_t B_t^{-1} g_t \, (3)$$

Where now we are preconditioning the gradient by any curvature matrix B. The focus of this project will be with Apollo and AdaHessian, although it is important to mention Hessian-free methods as it provides us a means to efficiently do calculations involving Hessian.

C. Methods:

In reviewing and discussing the methods, we focus on the central problem of empirical risk minimization in the supervised learning setting. Given a set of labeled data points $\{(x_1, y_1), ..., (x_m, y_m)\}$, a prediction model f parameterized by weights θ and a loss function L, we can define the empirical risk loss as,

$$E(\theta) = \sum_{i=1}^{m} L(f(x_i), y_i)$$

The reason for this focus is that the problem of minimizing empirical risk serves as the basis for all supervised learning algorithms and standard PAC theory says minimizing the risk for a large enough sample is sufficient for generalization over the unseen data generating distribution [7].

First, we briefly go over Newton's method to motivate the development of Apollo and AdaHessian. Newton's method attempts to minimize the empirical risk by constructing a sequence $\{\theta_t\}$ starting from an initial guess θ_0 using the following update

$$\theta_{t+1} = \theta_t - [\nabla^2 E]^{-1} \nabla E$$

Even though it enjoys a favorable theoretical convergence and often leads to better quality solutions faster, Newton's method suffers from a very large drawback which involves the Hessian matrix and its inverse. The first problem is that in certain applications, the analytic expression for Hessian is often complicated or intractable. Additionally, numerical methods for computing the second derivative also requires a lot of computations. Indeed, each Newton iteration costs $O(N^3)$

compared to $O(N)$ for each gradient descent iteration [8]. In addition to the prohibitive computational cost, Newton's method also requires $O(N^2)$ storage to store the Hessian compared to Gradient Descent's $O(N)$. With these disadvantages, it is impractical to apply Newton's method to applications with a large number of parameters like in the case training deep neural networks. This will always be the case in deep learning, which is why it is necessary to look at quasi-newton or hessian-less methods that work towards improving these computational complexities.

1. Apollo:
   As mentioned above, developing an efficient 2nd order optimization method that generalizes and performs well has been a topic of research in deep learning for some time now. The main problems with developing a quasi-newton optimization algorithm for deep learning revolves around the requirement of stochastic sampling. Since at every iteration a gradient is being estimated from a sample of the data, it can lead to an extremely noisy hessian.

   The first step in developing a quasi-newton method is to find a way to construct the curvature matrix B. As stated above, this needs to be both cheap and needs to take into account the noisy gradients that will construct this matrix. Computing a Diagonal approximation of the hessian has been found to be both efficient computationally and effective for improving convergence over first-order methods. Traditionally construction of the curvature matrix B follows from the secant equation, but the authors of Apollo found that this equation is not restrictive enough in a stochastic environment. As a result, the authors of Apollo develop a novel construction of the curvature matrix using parameter-wise updates based on the weak secant equation [10]. The weak secant equation is stated as follows:

$$\begin{aligned} B_{t+1} &= \underset{B}{\operatorname{argmin}} \|B - B_t\| \\ \text{s.t.} \quad & s_t^T B_{t+1} s_t = s_t^T y_t \quad \text{(weak secant equation)} \end{aligned}$$

   With $s_t = \theta_{t+1} - \theta_t$ and $y_t = g_{t+1} - g_t$ (where g is the stochastic gradient, and θ is simply the parameter to be optimized) In applications of deep neural networks with millions of parameters, it is not feasible to compute this update for every parameter. Thus they take advantage of the natural coupling of parameters in neural networks. This is done by decoupling our parameter θ into sets of parameters from each hidden layer l. This is described by the set:

$$\theta = \{\theta(l),\ l = 1, \dots, L\}$$

   Where L is the number of hidden layers. Of course, in deeper networks the cost of such an update will grow linearly. The benefits of constructing the curvature matrix using weak secant is that one only needs to explicitly compute the gradient. This is in essence what allows Apollo to maintain linear time and space complexity, although the coefficients of these complexities will be higher than that of first order methods. We will see the practical efficiency of such a formulation when compared with other methods used to construct B_t later (such as in the AdaHessian algorithm).

As we stated above, the current landscape of deep learning is dependent on stochastic sampling. The noise introduced by such a procedure is amplified when constructing the curvature matrix B, as we are using estimations of the gradient for its formulation at every iteration. For this reason, Apollo introduces a step-size bias correction step in order to circumvent this stochastic noise. They replace the stochastic gradient g_t by g'_t where $g'_t = \eta_t g_t$ and η_t is the optimal step size (1 if our B_t and g_t is close to the exact Hessian and gradient respectively). If the optimal step size is depicted to be 1, no change is made to our computations. Additionally, they also implement exponential moving averages for the stochastic gradient (momentum) to further circumvent the noisiness. As one may expect, it has been shown extensively that incorporating exponential moving averages for the stochastic gradients significantly reduces the variances [3]

To ensure that the matrix B_t is positive definite at every iteration t (as needed for guaranteed convergence), they employ what is called the rectified absolute value transformation:

$$D_t = \max(|B_t|, \omega).$$

Where $\omega > 0$, is called the convexity parameter and is found to be coupled with the optimal step size η_t parameter (under mild assumptions and with zero initialization of the algorithm). This coupling is important in reducing the hyper parameterization needed for proper behavior, and it is typical to fix $\omega = 1$ and computing η_t. Since B_t is a diagonal matrix, the cost of computing its absolute value is trivial.

These are the core computations behind Apollo, which provides its unique ability to cheaply construct a beneficial estimation of the Hessian at each iteration. The full details of every step in the algorithm can be found in the Apollo paper [6], as well as pseudocode for guiding implementation.

2. AdaHessian:
   Like Apollo, AdaHessian is a second order stochastic algorithm which dynamically incorporates the curvature of the loss function via adaptive estimates of the Hessian. In general, full Hessian methods are too computationally and spatially expensive whereas alternative second order methods that estimate the Hessian suffers from the noisiness of the estimates, thus destroying any possible benefits from including the Hessian. To address these shortcomings, AdaHessian introduces several novel approaches, namely: (i) a fast Hutchinson based method for curvature approximation with low overhead; (ii) a root-mean-square exponential moving average to smooth out variations of the Hessian diagonal and (iii) a block diagonal averaging to reduce the variance of Hessian diagonal elements.

   Let us first turn our attention to the adaptive first order methods, for which the general update formula is of the following form:

$$\theta_t = \theta_t - \alpha \frac{m_t}{v_t}$$

where m_t and v_t denote the first and second moment terms (momentum) at time t, respectively. From here, by specifying m_t and v_t, we arrive at multiple well-known first order optimizers. For example, Adagrad [1] specifies the following momentums:

$$m_t = \nabla E_t$$
$$v_t = \sqrt{\sum_{i=1}^{t} \nabla E_i \nabla E_i}$$

As another example, Adam, which has been considered the state-of-the-art optimizers for many NLP tasks, defines:

$$m_t = \frac{(1-\beta_1)\sum_{i=1}^{t} \beta_1^{t-i} \nabla E_i}{1-\beta_1^{t}}$$
$$v_t = \sqrt{\frac{(1-\beta_2)\sum_{i=1}^{t} \beta_2^{t-i} \nabla E_i \nabla E_i}{1-\beta_2^{t}}}$$

As can be observed, these methods, even though incorporate beneficial momentums to help with learning, only use information from the gradients. Motivated by the shown benefit of both momentums and Hessian, AdaHessian combines both idea and thus defines the following updates:

$$m^t = \frac{(1-\beta_1)\sum_{i=1}^{t} \beta_1^{t-i} \nabla E_i}{1-\beta_1^{t}}$$
$$v^t = \sqrt{\frac{(1-\beta_2)\sum_{i=1}^{t} \beta_2^{t-i} D_i D_i}{1-\beta_2^{t}}}$$

where D is a diagonal estimate of the Hessian. Thus, we've arrived at a new update where both momentums and curvature information are incorporated to guide the trajectory across the loss surface. The remaining question, however, is how do we compute D such that the memory and computation overhead is minimized while ensuring that D is a good diagonal estimate of the Hessian.

a. Hessian Diagonal Approximation:
One of the most common and simple Hessian approximation methods is to approximate the Hessian as a diagonal operator, $D = diag(H)$. There are multiple benefits to using a diagonal estimate. First, because D is a diagonal matrix, if $H \in R^{N \times N}$, there D will only have N non-zero elements, making the space complexity only $O(N)$, and thus making it feasible for models with millions of parameters like Neural Networks. Second, computing D as well as the inverse of D is much easier and much less computationally expensive. Finally, by using a Hessian diagonal estimate, we can efficiently perform spatial average (as shown later) as well as

computing its momentum, something which cannot be done for full Hessian. This allows us to smooth out noisy local curvature information and utilize the power of momentum, which has been shown to be very successful for first order methods.

The Hessian diagonal D can be computed using Hutchinson's method:

$$D = diag(H) = E[v \odot (Hv)]$$
$$\text{s.t } v \sim Rademacher(0.5)$$

Defined by the expression above, we need v to be a random vector with components sampled independently from a Rademacher distribution, which is a discrete distribution where a random variate has 50% chance of being +1 and 50% chance of being -1. As can be seen, the hessian H is still present in the expression. Thus, in order to **efficiently** compute the diagonal estimate, we borrow a crucial observation from Hessian Free method, where we observe that we can actually calculate the matrix-vector multiplication between the Hessian H and some vector v without having to explicitly form/compute H. More specifically, we can bypass storing and computing Hessian explicitly if we find a way to compute Hv for some random vector v. Indeed, we can work out that the i^{th} row of Hv is actually just the **directional derivative** of $\frac{\partial E}{\partial \theta_i}$ in the direction v [11]. This is crucial as we can approximate the directional derivative using finite differences. That is, let f be $\frac{\partial E}{\partial \theta_i}$, then,

$$\nabla_v\left(\frac{\partial E}{\partial \theta_i}\right) = \nabla_v f = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon v)-f(\theta)}{\varepsilon} \approx \frac{f(\theta+\varepsilon v)-f(\theta)}{\varepsilon}$$

Thus, we can then approximate the matrix-vector product Hv,

$$Hv \approx \frac{\nabla E(\theta+\varepsilon v)-\nabla E(\theta)}{\varepsilon}$$

With this efficient calculation of Hv, we now have an efficient expression to compute the Hessian diagonal estimate D. Indeed, with $v \sim Rademacher(0.5)$, Bekas et al. [9] proved that the expectation is a good estimate of actual Hessian diagonal.

b. Spatial Average:
One problem that may potentially hinder learning is that the Hessian diagonal can vary significantly for each parameter dimension of the problem. In other words, let us consider a single convolutional layer with 1 $N \times N$ filters, which means we have N^2 parameters in total. Thus, estimate D is then a $N^2 \times N^2$ diagonal matrix, or simply a N^2 - dimensional vector (since all elements other than the diagonal is 0). Yao et al. [5] pointed out that each of the N^2 parameters may have a very different corresponding value in D, and thus the entries in D will have significant variations. In order to smooth out the spatial variations, Yao et al. [5] performed spatial averaging for D, which is defined as:

$$D^{(s)}[ib+j] = \frac{\sum_{k=1}^{b} D[ib+k]}{b},\ 1 \le j \le b,\ 0 \le i \le \frac{N^2}{b} - 1$$

where N^2 is the total number of parameters as defined in our example above. $D \in R^{N^2}$ is the Hessian diagonal and $D^{(s)} \in R^{N^2}$ is the spatially averaged version of D. $D[i]$ $(D^{(s)}[i])$ refers to the i-th element of $D\,(D^{(s)})$ and b is the spatial average block size. In simpler term, all spatial averaging is doing is that it breaks the vector D into blocks of size b then proceeds to loop through each block, averaging the elements **inside each block** to arrive at $D^{(s)}$.

In addition to high level motivation, Yao et al. also showed that by replacing D by $D^{(s)}$, the update direction has the same convergence rate as using a full Hessian, assuming the function of interest is a simple strongly convex and strictly smooth function.

c. Hessian Momentum:
It has been shown that momentum is a very powerful tool for first order methods. Momentum, on a high level, is a moving average of a quantity of interest; in the first order case, it is the moving average of the gradients and thus helps with convergence rate as well as allowing optimizers to get out of local minima. With the same motivation, it is natural to extend momentum to the case of second order. Fortunately, we can easily apply momentum to the Hessian diagonal estimate D since it is a vector instead of a quadratically large matrix. More specifically, let D_t denote the Hessian diagonal with momentum, we have the following definition:

$$D_t = \sqrt{\frac{(1-\beta_2)\sum_{i=1}^{t} \beta_2^{\,t-i} D_i^{(s)} D_i^{(s)}}{1-\beta_2^{\,t}}}$$

where $D_i^{(s)}$ is the spatially averaged Hessian diagonal at time i and $0 < \beta_2 < 1$ is the second moment hyperparameter. We can observe that this is actually almost exactly the same as the second momentum term in Adam, with the only difference being that Hessian diagonal is used instead of gradient, which provides curvature information.

In summary, AdaHessian first uses Hutchinson's method to estimate the Hessian diagonal D, then uses spatial averaging across the diagonal and Hessian momentum to smooth out local variations in Hessian diagonal and to help with convergence rate. Thus, our AdaHessian update is defined as:

$$m_t = \frac{(1-\beta_1)\sum_{i=1}^{t} \beta_1^{\,t-i}\, \nabla E_i}{1-\beta_1^{\,t}}$$

$$v_t = \sqrt{\frac{(1-\beta_2)\sum_{i=1}^{t} \beta_2^{\,t-i}\, D_i^{(s)} D_i^{(s)}}{1-\beta_2^{\,t}}}$$

$$\theta_{t+1} = \theta_t - \alpha \frac{m_t}{v_t}$$

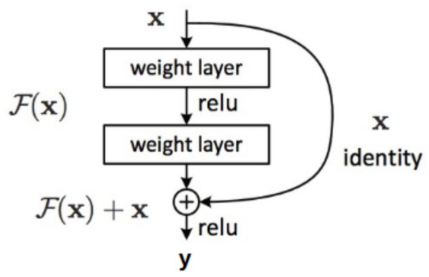
The main overhead of AdaHessian is the Hutchinson's method to approximate Hessian diagonal D. Yao et al. established a theoretical per-iteration cost of **2x** that of SGD. However, as will be discussed in section D, our experiment showed that AdaHessian is in fact significantly slower empirically, though still much faster and more feasible than the traditional second-order methods.

D. Results and Discussion:

a. Experimental setup:

Data. We tested AdaHessian and Apollo on two famous benchmark image classification datasets in the field of Computer Vision which is CIFAR-10 and STL10. CIFAR-10 consists of 60000 32x32 color images in 10 classes, yielding 6000 images per class. The 10 classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. The classes are completely mutually exclusive, meaning there's no overlap between any two classes. The dataset is divided into 50000 training images and 10000 testing images. STL-10 is pretty similar to CIFAR-10 but was intentionally made to be more challenging due to fewer training data and extracted from a broader distribution of images. STL-10 consists of only 12000 labelled images, each of size 92x92, which is significantly larger than that of CIFAR-10. Similar to CIFAR-10, there are the exact same 10 mutually exclusive classes. These datasets allow a consistent, repeatable environment to conduct experiments for deep learning.

Architecture. Our methods for comparing optimizers solely focused on the case of image classification through residual neural networks. Due to computational restrictions, the size of our inputs were relatively small, namely 32x32 and 92x92 images. These images get scaled (in the case of 92x92), and flattened into 1024 dimensional vectors. Residual networks [10], are a complex neural network architecture that cannot necessarily be detailed in this paper. However, it is important to note some specific attributes of residual networks and how they affect the backpropagation. Specifically, residual networks are depicted of blocks of layers known as residual blocks. Uniformly constructing groups of layers into blocks allows the backpropagation to add the input vector (x) to the resulting gradient computation. This in essence is a big part of the success residual networks have in eliminating the vanishing gradient problem. We illustrate the results of such a transformation, first the forward pass then the backward pass:


x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu
y

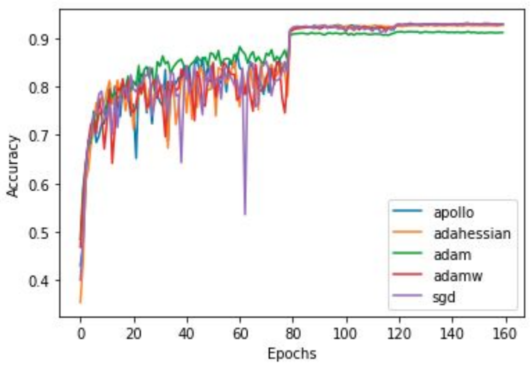
$$y = x + F(x)$$

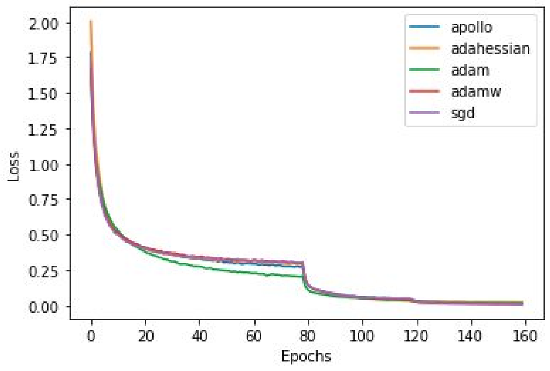
$$\frac{\delta E}{\delta x} = \frac{\delta E}{\delta y} * \frac{\delta y}{\delta x} = \frac{\delta E}{\delta y} * (1 + F'(x))$$

$$= \frac{\delta E}{\delta y} + \frac{\delta E}{\delta y} * F'(x)$$

For this project, we went with ResNet32 which is a very deep convolutional neural network of 32 residual blocks in total. We demonstrate that not only are Apollo and AdaHessian able to address the complexity issues and thus are practical in training this very deep network, but they also achieve very competitive performance as well as some other favorable properties that give them an edge over the first order methods.

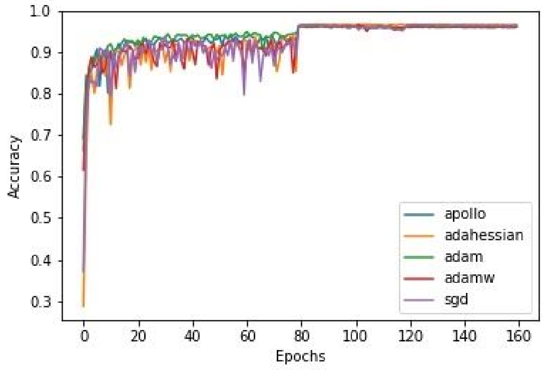
b. Result Figures and Tables:
CIFAR-10. The following are the accuracy and loss vs epochs graphs as well as a table of best performance and time taken per optimizer.

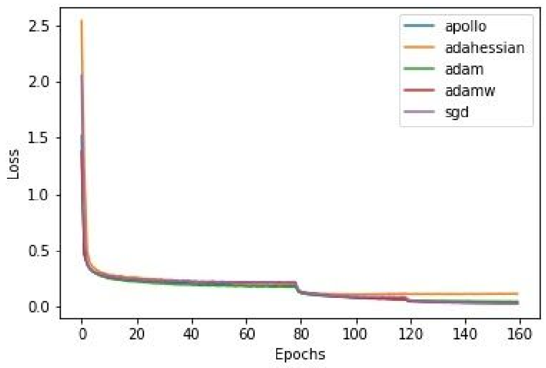




Optimizer	Apollo	AdaHessian	Adam	AdamW	S.G.D*
Accuracy	93.07	92.72	91.36	93.03	92.97
Time Elapsed	78.49	291.18	73.86	73.74	71.65

STL-10. We showcased the same figures and table for STL-10:





Optimizer	Apollo	AdaHessian	Adam	AdamW	S.G.D
Accuracy	96.60	96.65	96.19	96.47	96.51
Time Elapsed	112.34	434.82	107.10	106.48	103.53

c. Discussion:

Performance. Overall, we see slightly better optimums across both tasks for the second-order methods with varied added costs in training time. Since we have only focused on the specific task of image classification in this project, it is important to note that this pattern of performance repeats itself in the N.L.P (Natural Language Processing) experiments conducted in both Apollo [6] and AdaHessian papers [5]. Additionally, Yao et al. showed that AdaHessian is also better than S.O.T.A optimizer for Recommendation System tasks. This is **crucial** because this means both Apollo and AdaHessian are **consistently** comparable, if not better, to the S.O.T.A optimizer for each of the learning tasks. This is something that **cannot be said** for any of the first-order methods, because for example S.G.D is considered the best (S.O.T.A) optimizer for Computer Vision but performed much worse for N.L.P tasks, so in that sense its performance is not consistent across different tasks. This result is promising, but not conclusive to the general capabilities of the optimizers. As stated earlier, however, second-order methods by nature have very favorable properties that should lend themselves to be better in general across multiple tasks.

Complexity. From our experiments, we can clearly see that the cost of the Hutchinson's method of AdaHessian has a drastic effect on the practical run-time of the optimizer. This is a huge concern for the practicality of Adahessian, despite the generalization afforded by this way of constructing the curvature matrix. It remains to be seen the biggest cause of the inefficiency of the Hutchinson's method in practice, as theoretically the run-time should only be on the order of 2x that of first-order methods [5]. Thus, even though it's much faster than traditional second-order methods, it's still significantly slower than both Apollo and the first order methods. Indeed, our experiments showed that empirically, AdaHessian is ~4x slower than the remaining, across both datasets. Apollo, on the other hand, was able to benefit from the weak secant formulation of the curvature matrix whilst remaining comparable (in run-time) to the benchmark first-order methods selected. In addition to the slight improvements for image classification, Apollo boasts quite significant results in natural language processing [6].

Invariance. As a corollary, in practical deep learning settings, an important distinction to make about any optimizer is the sensitivity to the learning rate. One of the most favorable parts awarded by the nature of any second order method (the scaling of the gradient by the local curvature), is that learning rate sensitivity should be theoretically much lower than that of first order methods. In other words, second order methods, by nature, are much more **invariant** to changes in learning rate than first order methods. This is a crucial finding since this sensitivity to learning rate (or hyperparameters in general) is a common concern when trying to find the optimal value for large training tasks, which may take days to complete. Unfortunately, no experiments of this sort were able to be conducted for this report. However, Ma and Yao et al. showed in their papers that both AdaHessian and Apollo are indeed much more invariant to changes in learning rate than other first order methods like Adam. For example, Yao et al. compared Adam's and AdaHessian's performance on the Neural Translation task where the learning rate was varied between 8 different values. They found that Adam's performance, as expected, degraded rapidly as we moved away from the optimally tuned value whereas AdaHessian's performance remained consistently good (with only very slight decrease) across the 8 different learning rates. This alone gives the second order methods an important edge over the first order ones.

E. Conclusion:

Overall, this project explores and experimented with 2 recent second order optimization methods in detail. As our own verification results as well as the authors' results showed, these two methods are able to address both the time and space complexity issues with traditional second order methods, therefore being practical for training very deep neural nets, as well as yielding very competitive performance compared to the S.O.T.A first order optimizer for each learning task, from Computer Vision to N.L.P (Language Modeling, Neural Translation) to Recommendation System. Additionally, Apollo and AdaHessian also have very favorable properties, such as yielding consistently good performance across a wide spectrum of deep learning tasks (something that no first order methods can achieve) as well as being much more invariant to changes in learning rate. All this combined gives Apollo and AdaHessian an edge over the first order methods.

F. References:

[1] J. Duchi, E. Hazan and Y. Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. The Journal of Machine Learning Research. July, 2011.

[2] M. D. Zeiler. Adadelta: an Adaptive Learning Rate Method. Retrieved from: https://arxiv.org/pdf/1212.5701.pdf

[3] D. P. Kingma and J.L. Ba. Adam: A Method for Stochastic Optimization. In International Conference on Learning Representations. 2015.

[4] D. Kovalev, K. Mishchenko and P. Richtarik. Stochastic Newton and Cubic Newton Methods with Simple Local Linear-Quadratics Rates. In Neural Information Processing Systems (NeurIPS). 2019.

[5] Z. Yao, A. Gholami, S. Shen, K. Keutzer and M. W. Mahoney. AdaHessian: An Adaptive Second Order Optimizer for Machine Learning. 2020.
Retrieved from: https://arxiv.org/pdf/2006.00719.pdf

[6] X. Ma. Apollo: an Adaptive Parameter-wise Diagonal Quasi-Newton Method for Nonconvex Stochastic Optimization. 2020.
Retrieved from: https://arxiv.org/pdf/2009.13586.pdf

[7] N. Agarwal. Second Order Optimization Methods for Machine Learning. Princeton University Dissertations. 2018.

[8] R. Tibshirani. Newton's Method. Retrieved from: http://www.stat.cmu.edu/~ryantibs/convexopt-F15/lectures/14-newton.pdf

[9] C. Bekas, E. Kokiopoulou, and Y. Saad. An Estimator for the Diagonal of a Matrix. Applied Numerical Mathematics. 2007.

[10] John E Dennis, Jr and Henry Wolkowicz. Sizing and least-change secant methods. SIAM Journal on Numerical Analysis, 30(5):1291–1314, 1993.

[11]Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. (2015). Deep Residual Learning for Image Recognition.