Natural Gradient Method: Perspectives, Efficient-Scalable Approximation, and Analysis

Rajesh Shrestha
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR 97331
shresthr@oregonstate.edu

Abstract

Natural Gradient Descent, a second-degree optimization method motivated by the information geometry, makes use of the Fisher Information Matrix instead of the Hessian which is typically used. However, in many cases, the Fisher Information Matrix is equivalent to the Generalized Gauss-Newton Method, that both approximate the Hessian. It is an appealing method to be used as an alternative to stochastic gradient descent, potentially leading to faster convergence. However, being a second-order method makes it infeasible to be used directly in problems with a huge number of parameters and data. This is evident from the community of deep learning sticking with the stochastic gradient descent method since the beginning. In this paper, we look at the different perspectives on the natural gradient method, study the current developments on its efficient-scalable empirical approximations, and finally examine their performance with extensive experiments.

1 Introduction

The gradient descent method Kiefer and Wolfowitz [1952], Robbins and Monro [1951] has been very popular in non-linear optimization including the field of machine learning and deep learning. The stochastic version i.e. Stochastic Gradient Descent method (SGD) is a simple and light method that scales very well to train big deep learning models on a vast amount of data. Almost all the work in the field of deep learning has used SGD in some form for training the model. This has been used in all domains including computer vision, reinforcement learning, natural language processing, etc. in tasks like face recognition Hu et al. [2015], object recognition Eitel et al. [2015], control Huang et al. [2019] and much more. This first-order optimization method utilizes the empirical estimation of the gradient in the parameter space. There have been multiple variations to this method like Nesterov accelerated gradient Nesterov [1983], SGD with momentum Qian [1999], RMS prop Hinton et al. [2012], Adam Kingma and Ba [2014], etc. that improves its stability and convergence. However, the performance of this method is limited being a first-order method. There has been much interest in using the second-order optimization method to improve the convergence and reduce the training time Becker and Le Cun [1988], Grosse and Martens [2016], Osawa et al. [2020], and the natural gradient descent method Grosse and Martens [2016], Osawa et al. [2020], Zhang et al. [2019] has grasped the attention of the community.

As the name implies, the natural gradient method Amari [1998] uses the natural gradient instead of the standard gradient which is defined as the product of the inverse of the Fisher Information matrix(FIM) and the gradient. This method has been motivated to take the steepest descent in the space of realizable distribution rather than the space of parameters, where the "Riemannian metric" is used for computing distance in the distribution Amari and Nagaoka [2000]. This metric approximates the square root of KL divergence locally and is only dependent on the distribution itself, not the

parameterization. From this perspective of natural gradient, this method is invariant to any smooth and invertible reparameterization of the model Ollivier [2013], Martens [2020].

The natural gradient method, being closely related to the Gauss-Newton method, seems to require a much less number of iterations than SGD making it a potential and appealing method. But this doesn't always translate to faster convergence in practice. Amari [1998] argued that this is contingent on how well the cost function being optimized is well approximated by a convex quadratic. The training of neural networks involves optimizing the non-convex and potentially non-smooth loss functions, making it difficult to prove any form of convergence guarantee or bound. In addition, the large size of the Fisher Information Matrix in the case of highly parameterized neural networks makes it infeasible to be used. This problem can be mitigated through approximation of FIM such that the computation, storage, and inversion of this matrix is done efficiently Grosse and Martens [2016], George et al. [2018], Goldfarb et al. [2020], Soori et al. [2021]. In this paper, we study the theoretical definition, and interpretation along with various efficient-scalable versions of the natural gradient method and compare them with SGD to study its properties through empirical study.

In the following section, we first define the Fisher Information matrix and derive its relationship with the Hessian matrix. After it, we will define natural gradient and discuss its geometric interpretation leading to the derivation of natural gradient methods. Finally, we will discuss some of the approximate natural gradient methods and analyze them through experiments.

2 Background

2.1 Fisher Information Matrix(FIM)

Definition: Suppose we have a model parameterized by θ that models the distribution $p(x|\theta)$. Then, the likelihood of the parameter θ is given by equation 1 and similarly, the loglikelihood is given by equation 2.

$$L(\theta|x) = p(x|\theta) \tag{1}$$
$$l(\theta|x) = log(L(\theta|x)) \tag{2}$$
$$= log(p(x|\theta))$$

The parameters are usually estimated by maximizing this loglikelihood function in the frequentist approach. Hence, the negative of this loglikelihood is often referred to as loss/cost in training. The necessary condition for this estimated θ to be an optimal one is that this θ point should correspond to one of the stationary points in the likelihood function. Hence, we use the gradient of likelihood function defined as a score function (equation 3) to analyze the estimated value.

$$s(\theta|x) = \nabla_\theta l(\theta|x) \tag{3}$$

Then, the expected value of this score function is 0 (Appendix A.1).

$$\therefore \mathbb{E}_{p(x|\theta)}[s(\theta|x)] = 0 \tag{4}$$

Then, the fisher information matrix (FIM) is basically the expected measure of uncertainty in the $\mathbf{s}(\theta|\mathbf{x})$ *around the expected value i.e. covariance matrix of the score function (equation 5) and contains the curvature information.* For simplicity, we denote the score function by only s below.

$$\mathbf{F}(\theta) = \mathbb{E}_{p(x|\theta)}[(s - \mathbb{E}_{p(x|\theta)}[s])(s - \mathbb{E}_{p(x|\theta)}[s])^T] \tag{5}$$
$$= \mathbb{E}_{p(x|\theta)}[\nabla_\theta l(\theta|x).\nabla_\theta l(\theta|x)^T] \tag{6}$$

Usually, it's not feasible to compute the expectation in equation 6, especially in the neural networks as the distribution is very complicated and most likely very high dimensional. In such cases, the expectation is intractable. It's common for it to be estimated using the data distribution $q(x)$ instead. Given the data sample of size N, the empirical estimation of the FIM is given by

$$\mathbf{F}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta logp(x_i|\theta)\nabla_\theta logp(x_i|\theta)^T \tag{7}$$

Relation between FIM and Hessian: *The Fisher Information Matrix (FIM) is equal to the negative of the expected value of the hessian of loglikelihood as shown in equation 8(Appendix A.2) and measures the curvature information which is further explained in section 2.2.1*

$$\therefore \mathbf{F}(\theta) = -\mathbb{E}_{p(x|\theta)}[\mathbf{H}(l(\theta|x))] \tag{8}$$

2.2 Natural Gradient

Let a model parameterized by θ as before that models the distribution $p(x|\theta)$. Let $\mathbb{X}$ denote the sample data of size N and $\mathcal{L}(\theta|\mathbb{X})$ denotes the loss function i.e. negative loglikelihood function (equation 10).

Simply, the natural gradient is the product of the inverse of FIM and the gradient as shown in equation 9. In the following subsections, we examine the geometric interpretation of this natural gradient and show how this quantity is derived Martens [2020].

$$\tilde{\nabla}_\theta = F(\theta)^{-1}\nabla_\theta\mathcal{L} \tag{9}$$

$$\mathcal{L}(\theta|\mathbb{X}) = -L(\theta|\mathbb{X}) \tag{10}$$

2.2.1 Distribution Space:

The update equation for the original gradient descent uses the gradient of the loss function $\mathcal{L}$ to update in the parameter space.

$$\theta^+ = \theta - \alpha\nabla_\theta\mathcal{L}(\theta|\mathbb{X}) \text{ ,where } \alpha \text{ is learning rate} \tag{11}$$

This method selects the direction on the θ space so that the loss $\mathcal{L}$ decreases the most i.e. highest reduction in loss with a unit change in the parameter θ. Mathematically,

$$-\frac{\nabla_\theta\mathcal{L}}{||\nabla_\theta\mathcal{L}||} = \lim_{\epsilon \to 0} \frac{1}{\epsilon} \arg\min_{t:||t||\leq\epsilon} \mathcal{L}(\theta + t) \tag{12}$$

The equation 12 shows that the steepest direction is the one that is within ϵ proximity such that $\mathcal{L}(\theta+t)$ is the minimum obtainable with the constraint. We can see that the neighborhood is defined by $||t||$ which is dependent on the euclidean space of parameters θ, as a result, this method is dependent on the euclidean geometry of parameter space.

Let's suppose, the actual underlying distribution of the data generation process is $q(x)$ which isn't available to us. Then our aim is to learn θ so that the distance between the two distributions – distribution with learned model $p(x|\theta)$ and the underlying distribution $q(x)$ – is smaller. *Hence, it is more logical and sensible to take a step in the direction of descent in this distance in the model's distribution space instead of euclidean parameter space. Intuitively, taking even a small step in parameter space could result in a drastic change in the distribution so a step in the realizable distribution space should be taken instead.* One form of measure of distance is the KL divergence.

Relation between KL divergence and Fisher Information Matrix: *FIM is the hessian of KL-divergence of two distributions $p(x|\theta)$ and $p(x|\theta')$ evaluated at $\theta' = \theta$(Appendix A.3). FIM defines the curvature of the distribution space using the KL divergence as the metric.*

$$\mathbf{F}(\theta) = \nabla^2_{\theta'}\text{KL}(p(x|\theta)||p(x|\theta'))|_{\theta'=\theta} \tag{13}$$

2.3 Natural Gradient Descent(NGD): Steepest Descent in the distribution space

Let's approximate the KL divergence between two distributions with parameters θ and θ' such that $t = \theta' - \theta$. In the following for simplicity, we use the notation p_θ to denote $p(x|\theta)$. Using second-order Taylor expansion,

$$\text{KL}(p_\theta||p_{\theta'}) \approx \text{KL}(p_\theta||p_{\theta'})|_{\theta'=\theta} + t(\nabla_{\theta'}\text{KL}(p_\theta||p_{\theta'}))|_{\theta'=\theta} + \frac{1}{2}t^T(\nabla^2_{\theta'}\text{KL}(p_\theta||p_{\theta'}))|_{\theta'=\theta}t$$

$$= 0 - t\mathbb{E}_{p_\theta}(\nabla_{\theta'}log(p_\theta)) + \frac{1}{2}t^T F(\theta)t$$

$$= 0 - 0 + \frac{1}{2}t^T F(\theta)t$$

$$\therefore \text{KL}(p_\theta||p_{\theta'}) \approx \frac{1}{2}t^T F(\theta)t \tag{14}$$

Now, let's formalize the problem of finding the steepest descent in the distribution space. Let t^* be the direction corresponding to the steepest descent similar to equation 12, then,

$$t^* = \arg\min_{t \text{ s.t. } \text{KL}(p_\theta||p_{\theta+t})<=\epsilon^2} \mathcal{L}(\theta + t) \tag{15}$$

3

Using Lagrange Multiplier, we get

$$t^*, \lambda^* = \arg\max_{\lambda} \arg\min_{t} \mathcal{L}(\theta + t) + \lambda(KL(p_\theta || p_{\theta+t}) - \epsilon^2) \tag{16}$$

$$\approx \arg\max_{\lambda} \arg\min_{t} \mathcal{L}(\theta) + t\nabla_\theta \mathcal{L}(\theta) + \lambda\left(\frac{1}{2}t^T F(\theta)t - \epsilon^2\right) \tag{17}$$

Taking the first derivatives we get,

$$\nabla_\theta \mathcal{L}(\theta) + \lambda F(\theta)t = 0$$

$$t^* = -\frac{1}{\lambda^*} F(\theta)^{-1} \nabla_\theta \mathcal{L}(\theta) \tag{18}$$

By dual feasibility, the value of $\lambda^* >= 0$, hence the direction of steepest descent is given by $-F(\theta)^{-1}\nabla_\theta \mathcal{L}(\theta)$ which is same as the negative of quantity that we defined as the natural gradient in equation 9. The constant positive term can be absorbed in the learning rate α.

Algorithm 1 Natural Gradient Descent Method

1: **Input**: data $\mathbb{X}$, learning rate α
2: Initialize model parameters θ
3: **while** not convergence in θ **do**
4:     Sample data and forward pass in the model
5:     Compute the loss $\mathcal{L}$ for the passed data
6:     Compute the gradient $\nabla_\theta \mathcal{L}$
7:     Compute FIM or its empirical approximation:
8:         $\mathbf{F}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta log p(x_i|\theta)\nabla_\theta log p(x_i|\theta)^T$
9:     Compute the natural gradient: $\tilde{\nabla}_\theta = F^{-1}\nabla_\theta \mathcal{L}(\theta)$
10:     Update Model parameter: $\theta^+ = \theta - \alpha\tilde{\nabla}_\theta$
11: **end while**

---

3 Efficient-Scalable Approximations

Despite having strong theoretical significance and potential of natural gradient, this method in its original form is limited in the scope of models to which it can be applied. Most of the neural networks that are used in practice have a very high number of parameters that makes the use of this method in them infeasible as a very high dimensional FIM matrix needs to be computed, stored, and inverted too. The number of elements in FIM scales quadratically to the number of parameters in a model. This is the main reason why simple first-order gradient methods have been popular over these better second-degree methods.

Suppose we have a deep neural network f with L layers parameterized by θ with p number of parameters in total. Let θ^l be the parameters corresponding to each layer of the network. Let d_i^l be the input size and d_o^l be the output size of the lth layer. With the use of exact FIM, the storage required is $O(p^2)$ and the computation of the inverse of FIM for natural gradient (equation 9) requires $O(p^3)$. To mitigate this problem, most of the natural gradient methods approximate FIM using a block-diagonal matrix such that the elements corresponding to cross layers are 0 as shown in figure 1. The problem reduces to finding the FIM corresponding to each layer i.e. F_l for $l = 1, ..., L$.

Even with this block-diagonal approximation, the requirement in storage and computation is still high for layers that have a high number of parameters, for instance, a fully connected layer with high input and output dimensions, the requirement in storage and computation is high. To mitigate this issue, the inverse of block FIM is further approximated using efficient and smaller matrices. KFAC Martens and Grosse [2015], Grosse and Martens [2016] uses the Kronecker product of two small matrices to approximate the inverse of FIM for a layer. The inverse can still be expensive for wide layers so EKFAC George et al. [2018] improves this approximation by rescaling the Kronecker factors using a diagonal matrix which is obtained by using svd. In contrast, EBFGS approximates the inverse of the Kronecker factors using low-rank BFGS updates. Among all the development, a recent method TENGraD Soori et al. [2021] performs better in terms of approximation accuracy, time efficiency, and storage requirement. This method performs exact FIM block inversion by using computationally
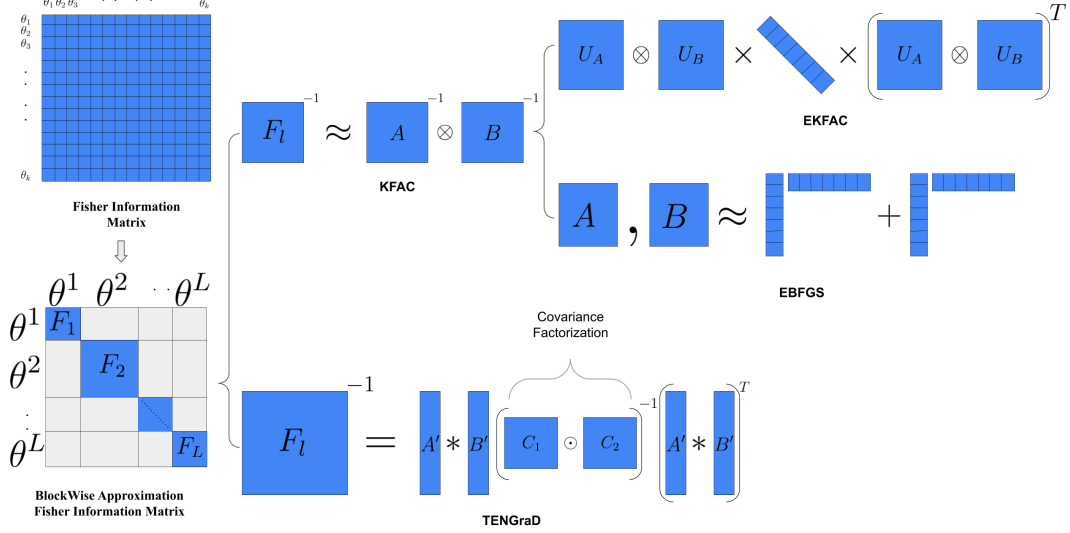
Figure 1: Approximations of FIM in natural gradient method Soori et al. [2021]

efficient covariance factorization and reusing the intermediate vectors and matrices. Further details on TENGraD can be found in Appendix A.5. In the following sections, we perform experiments using these methods and analyze the results obtained.

4   Experiments

Apart from the theoretical interpretation and claims, I performed experiments to see how much those things hold in practice and how the different approximations compare to the gradient-based method SGD. The experiment was performed in the deep neural network(DNN) in different settings and using different datasets. The set of experiments performed here can be extended to other architectures of neural network including but not limited to Convolutional Neural Networks (CNN). While working with the deep models in this experiment, TENGraD is mostly used as a natural gradient method as other methods aren't scalable to models with a huge number of parameters. In addition, the TENGraD method is equivalent to a block-wise natural gradient but an efficient one. My analysis targets three properties that are important to evaluate any optimization method: *Convergence*, *Performance*, *Stability*, and *Scalability*.

4.1   Convergence

Being the second-degree method, the natural gradient methods (both exact NGD and TENGraD) do often converge very faster in practice, even with the empirical FIM matrix. Exact NGD method seems to perform better than the approximation-based method which is expected. From figure 2, it's clear that the natural gradient method converges faster than the gradient-based method(SGD). This is not only due to the effective learning rate α induced by the Fisher Information Matrix(FIM) which can be seen with the grid search for lr in figure 2b. Even with using the best αs, the natural gradient method(NGD) still performs better than SGD. We can still see similar behavior in deep and wide networks given that the algorithms are made stable for these models.

This convergence property seems to hold for deep models with wide layers too (figure 3).

Both SGD and NGD use a mini-batch of data to estimate the gradient and hessian locally. This estimate will be noisy on using a sub-sample of data and will impact the convergence of both methods. However, NGD uses both first-order and second-order derivatives estimated from data to update so will be impacted more by the batch size. Our result (figure 4) exactly matched this speculation. We can observe that the NGD methods converge very faster compared to SGD methods when the batch
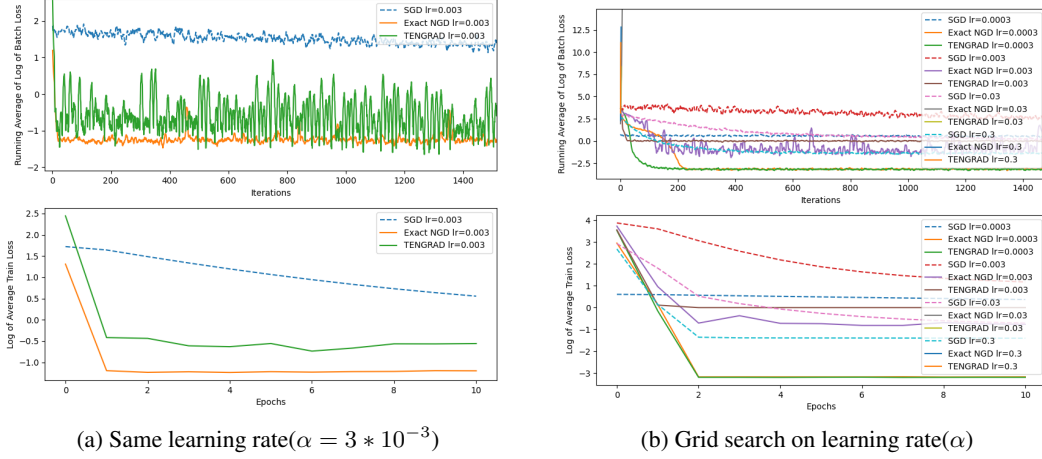
(a) Same learning rate($\alpha = 3 * 10^{-3}$)　　　　　(b) Grid search on learning rate(α)

Figure 2: Convergence in DNN using weather dataset for all methods(with one hidden layer of size 4) and batch size=128

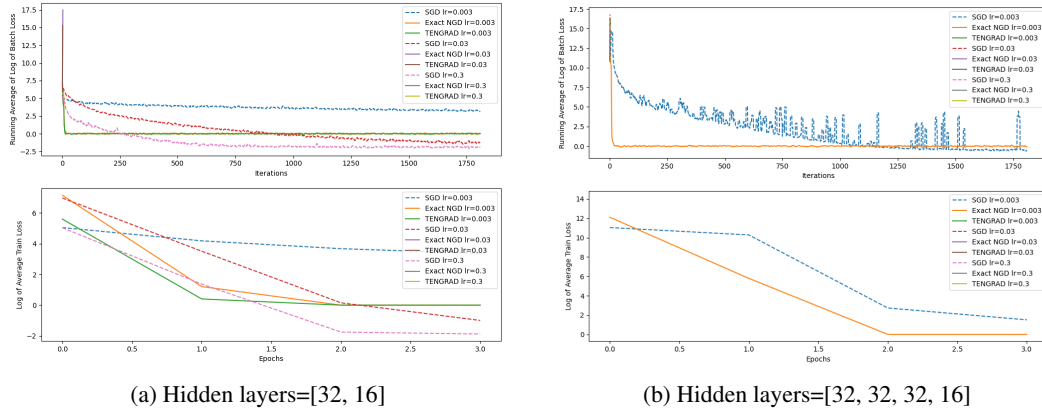(a) Hidden layers=[32, 16]　　　　　(b) Hidden layers=[32, 32, 32, 16]

Figure 3: Convergence in Deep and/or Wide DNN using grid lr search and weather dataset with batch size=128

size is high but the convergence significantly worsens when the batch size is reduced. Hence, the NGD methods are highly dependent on batch size.

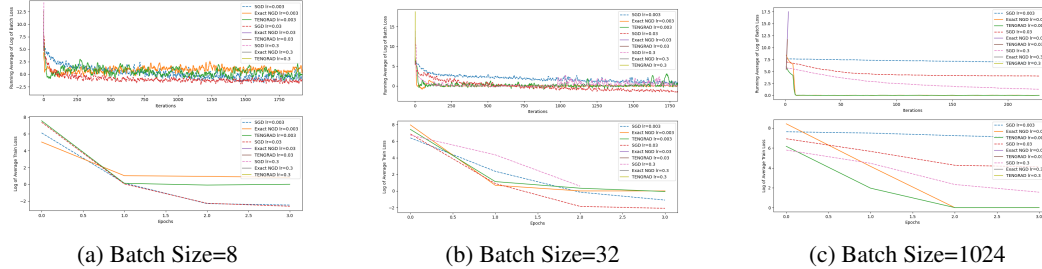(a) Batch Size=8　　　　　(b) Batch Size=32　　　　　(c) Batch Size=1024

Figure 4: Impact of batch size in convergence. DNN with hidden layers=[32,16] in weather dataset

4.2 Performance

Figure 2, 3, and 4 shows that in certain situations NGD methods often converge faster than SGD. However, does that translate to actual better performance in terms of the objective value at convergence

Table 1: Log of Training Loss at Convergence using batch size $m = 128$

Hidden Layers	Method	Weather	House Price	Ecoli
4	SGD	-0.45	-0.192	**-1.96**
	TENGraD	**-2.96**	**-1.120**	-0.16
32,16	SGD	-1	-0.794	**-1.37**
	TENGraD	**-3.68**	**-0.876**	0.59
32,16,16	SGD	0.24	-0.16	0.39
	TENGraD	**-1.65**	**-1.06**	**0.25**

i.e. convergence to a better or as good value as SGD method? In this subsection, I will try to answer these questions based on the experiments.

The argument Amari [1998] that NGD methods perform better when the loss function can be well approximated by a convex quadratic aligns with the results that is obtained from the experiment. The first important factor is the batch size itself as discussed in subsection 4.1. The loss function is estimated with data of batch size that keeps changing based on the data sample itself and this variance is highly dependent on the sample size. From another perspective, the performance is dependent on the reliability of the empirical FIM matrix itself which is dependent on the batch size used to estimate it. From figure 4, it's evident that the loss at convergence for NGD methods is better for higher values of batch size but this performance degrades along with convergence with reduced batch size. Another factor seems to be the complexity of the model. The performance seems to degrade as models get complex and as a result, the loss function curvature gets complex as seen in figure 2 and 3. Exact NGD methods are not feasible for deep neural networks, so we compare the loss at convergence for TENGraD and SGD methods. For this experiment, we use a batch size=128 that is around the value that is usually used in the deep learning community. We used three datasets for this: Weather [1](Regression), House Price[2](Regression) and Ecoli [3](Classification). TENGraD method seems to work better in the least square loss function compared to the cross-entropy loss function(Table 1).

4.3 Stability

The update using NGD is comparatively very noisy. The low batch size and the high number of parameters exacerbate this problem. It's very common for the FIM formed by the equation 7 to be ill-conditioned. It's very likely that the empirical FIM matrix has either very large eigenvalues or very small eigenvalues, both resulting in the instability of the NGD methods. From equation 7, we can see that FIM is very likely to be rank deficient. Upon taking inverse of FIM, the inverse matrix either blows up or zeros the update direction obtained using natural gradient(equation 9). In addition, a stochastic empirical approximation of FIM makes it even worse. This is the reason why we added a constant βI in equation 36. Adding a small value of β prevents the condition of FIM being a singular matrix but the inverse would give a natural gradient having very huge matrices. However, if we add a bigger value of β, we can circumvent this problem, but this would defeat the purpose of using curvature-aware FIM and will be a scaled version of SGD itself for a high value of β. In the experiment, this instability issue seems to be more serious in a model trained for a classification task with cross-entropy loss function. We also found NGD methods to be very sensitive to the learning rate α, so using a decaying value of it and a regularization in the weights of the network helped to stabilize the training of the NGD methods. Occasionally, the model could even diverge if continued training after converging to a fixed value. The decay learning rate also helped in this.
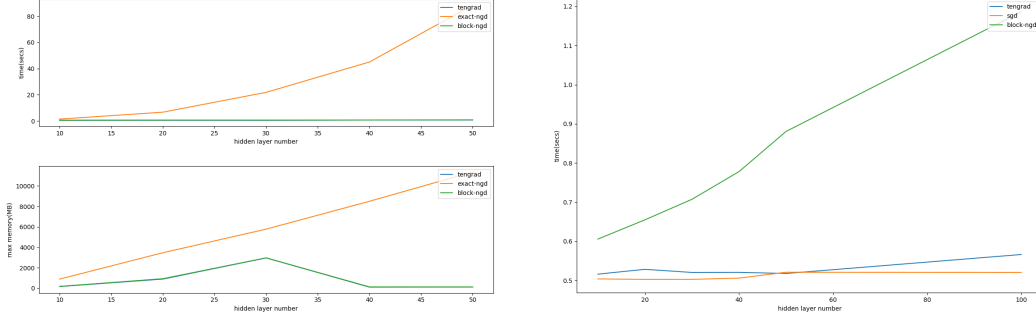
4.4 Scalability

The NGD method in its original form scales $O(n^2)$ in space and $O(n^3)$ in time complexity if n is the total number of parameters of a model. The approximation methods alleviate this problem with certain assumptions. Blockwise NGD assumes the FIM to be block diagonal and TENGraD further

---

[1]Weather dataset: https://www.kaggle.com/datasets/muthuj7/weather-dataset
[2]House Price dataset: https://www.kaggle.com/datasets/harlfoxem/housesalesprediction
[3]Ecoli dataset: Dua and Graff [2017]

(a) Time and memory requirement for Exact NGD, Blockwise NGD and TENGraD

(b) Time comparison between SGD, Blockwise NGD and TENGraD

Figure 5: Comparison of memory and time requirements in a model with each hidden layer has 20 nodes

uses Woodbury Matrix Identity to change the inversion of the block-wise FIM matrix into consecutive time-efficient matrix operations. The efficiency in time and space requirements obtained from them in comparison to the original NGD can be clearly seen in figure 5a. The time and space requirement for Exact NGD grows so fast making it infeasible for models with large parameters like deep neural network. The computation requirement in TENGraD is drastically reduced from the Blockwise NGD and scales in a similar fashion to the SGD method (figure 5b).

5 Conclusion

With the new approximation method like TENGraD, the natural gradient method can easily scale even for deep learning models with a very high number of parameters. In certain conditions, NGD methods can definitely converge faster and reduce the number of iterations required. This will inturn lead to a reduction in time, as the time requirement for TENGraD scales similar to SGD. It also leads to better performance if the loss function can be approximated well enough by a convex quadrature. Theoretically, it has such an elegant interpretation as a steep descent in the distribution space, however, in practice, the stability issues and the hypersensitivity with respect to the hyperparameters make it less appealing.

The methods like RMSprop and Adam also maintain the running second-order moments approximating the FIM by considering it to be a diagonal matrix that makes the computation, storing, and inverting to O(n). They are relatively stable and work well with deep-learning models. I believe NGD can have a huge impact on deep learning and optimization but not with its current nascent state. Further research needs to be done on making NGD methods stable and perform at least as well as the established first-order methods. Adding momentum to the NGD methods might help and is reserved for future work.

References

Shun-ichi Amari. Natural Gradient Works Efficiently in Learning. *Neural Computation*, 10(2): 251–276, 02 1998. ISSN 0899-7667. doi: 10.1162/089976698300017746. URL `https://doi.org/10.1162/089976698300017746`.

Shun-ichi Amari and Hiroshi Nagaoka. *Methods of information geometry*, volume 191. American Mathematical Soc., 2000.

Sue Becker and Yann Le Cun. Improving the convergence of back-propagation learning with. 1988.

Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL `http://archive.ics.uci.edu/ml`.

Andreas Eitel, Jost Tobias Springenberg, Luciano Spinello, Martin Riedmiller, and Wolfram Burgard. Multimodal deep learning for robust rgb-d object recognition. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 681–687. IEEE, 2015.

Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast approximate natural gradient descent in a kronecker factored eigenbasis. *Advances in Neural Information Processing Systems*, 31, 2018.

Donald Goldfarb, Yi Ren, and Achraf Bahamou. Practical quasi-newton methods for training deep neural networks. *Advances in Neural Information Processing Systems*, 33:2386–2396, 2020.

Roger Grosse and James Martens. A kronecker-factored approximate fisher matrix for convolution layers. In *International Conference on Machine Learning*, pages 573–582. PMLR, 2016.

Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14(8):2, 2012.

Guosheng Hu, Yongxin Yang, Dong Yi, Josef Kittler, William Christmas, Stan Z Li, and Timothy Hospedales. When face recognition meets with deep learning: an evaluation of convolutional neural networks for face recognition. In *Proceedings of the IEEE international conference on computer vision workshops*, pages 142–150, 2015.

Qiuhua Huang, Renke Huang, Weituo Hao, Jie Tan, Rui Fan, and Zhenyu Huang. Adaptive power system emergency control using deep reinforcement learning. *IEEE Transactions on Smart Grid*, 11(2):1171–1182, 2019.

J. Kiefer and J. Wolfowitz. Stochastic Estimation of the Maximum of a Regression Function. *The Annals of Mathematical Statistics*, 23(3):462 – 466, 1952. doi: 10.1214/aoms/1177729392. URL `https://doi.org/10.1214/aoms/1177729392`.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Agustinus Kristiadi. Natural gradient descent. URL `https://agustinus.kristia.de/techblog/2018/03/14/natural-gradient/`.

James Martens. New insights and perspectives on the natural gradient method. *The Journal of Machine Learning Research*, 21(1):5776–5851, 2020.

James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015.

Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o $(1/k^2)$. In *Doklady an ussr*, volume 269, pages 543–547, 1983.

Yann Ollivier. Riemannian metrics for neural networks. *Information and Inference: a Journal of the IMA*, 2, 2013.

Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Chuan-Sheng Foo, and Rio Yokota. Scalable and practical natural gradient for large-scale deep learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1): 145–151, 1999. ISSN 0893-6080. doi: https://doi.org/10.1016/S0893-6080(98)00116-6. URL `https://www.sciencedirect.com/science/article/pii/S0893608098001166`.

Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400 – 407, 1951. doi: 10.1214/aoms/1177729586. URL `https://doi.org/10.1214/aoms/1177729586`.

Saeed Soori, Bugra Can, Baourun Mu, Mert Gürbüzbalaban, and Maryam Mehri Dehnavi. Tengrad: Time-efficient natural gradient descent with exact fisher-block inversion. *arXiv preprint arXiv:2106.03947*, 2021.

Guodong Zhang, James Martens, and Roger B Grosse. Fast convergence of natural gradient descent for over-parameterized neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

A Appendix

A.1 Expectation of score function Kristiadi

Claim: The expected value of the score function is 0.

$$
\begin{aligned}
\mathbb{E}_{p(x|\theta)}[s(\theta|x)] &= \mathbb{E}_{p(x|\theta)}[\nabla_\theta l(\theta|x)] \\
&= \int_x p(x|\theta)\nabla_\theta l(\theta|x)dx \\
&= \int_x p(x|\theta)\frac{\nabla_\theta p(x|\theta)}{p(x|\theta)}dx \\
&= \int_x \nabla_\theta p(x|\theta)dx \\
&= \nabla_\theta \int_x p(x|\theta)dx \\
&= \nabla_\theta 1 \\
\therefore \mathbb{E}_{p(x|\theta)}[s(\theta|x)] &= 0
\end{aligned}
\tag{19}
$$

A.2 Relationship between Hessian and FIM Kristiadi

The hessian(**H**) of the model $f(x|\theta)$ is the second-order partial derivative of the $l(\theta|x)$. This is the same as the Jacobian(**J**) of the gradient of $l(\theta|x)$.

$$
\begin{aligned}
\mathbf{H}(l(\theta|x)) &= \mathbf{J}(\nabla_\theta l(\theta|x)) \\
&= \mathbf{J}(\frac{\nabla_\theta p(x|\theta)}{p(x|\theta)}) \\
&= \frac{p(x|\theta)\mathbf{H}(p(x|\theta)) - \nabla_\theta p(x|\theta)\nabla_\theta p(x|\theta)^T}{p(x|\theta)^2} \\
&= \frac{\mathbf{H}(p(x|\theta))}{p(x|\theta)} - \frac{\nabla_\theta p(x|\theta)\nabla_\theta p(x|\theta)^T}{p(x|\theta)^2} \\
&= \frac{\mathbf{H}(p(x|\theta))}{p(x|\theta)} - \nabla_\theta l(\theta|x)\nabla_\theta l(\theta|x)^T
\end{aligned}
\tag{20}
$$

$$
\begin{aligned}
\mathbb{E}_{p(x|\theta)}[\mathbf{H}(l(\theta|x))] &= \mathbb{E}_{p(x|\theta)}\left[\frac{\mathbf{H}(p(x|\theta))}{p(x|\theta)}\right] - \mathbb{E}_{p(x|\theta)}[\nabla_\theta l(\theta|x)\nabla_\theta l(\theta|x)^T] \\
&= \int_x p(x|\theta)\frac{\mathbf{H}(p(x|\theta))}{p(x|\theta)}dx - \mathbf{F}(\theta) \\
&= \int_x \mathbf{H}(p(x|\theta))dx - \mathbf{F}(\theta) \\
&= \mathbf{H}(\int_x p(x|\theta)dx) - \mathbf{F}(\theta) \\
&= \mathbf{H}(1) - \mathbf{F}(\theta) \\
&= 0 - \mathbf{F}(\theta) \\
\therefore \mathbf{F}(\theta) &= -\mathbb{E}_{p(x|\theta)}[\mathbf{H}(l(\theta|x))]
\end{aligned}
\tag{21}
$$

A.3 Relationship between FIM and KL divergence Kristiadi

Let θ and θ' be two instances of the parameter of the model. Then, the KL divergence between the model's distribution corresponding to these two instantiations is $\mathrm{KL}(p(x|\theta)||p(x|\theta'))$.

$$\text{KL}(p(x|\theta)||p(x|\theta')) = \mathbb{E}_{p(x|\theta)}\left[log\left(\frac{p(x|\theta)}{p(x|\theta')}\right)\right]$$

$$= \mathbb{E}_{p(x|\theta)}[log(p(x|\theta))] - \mathbb{E}_{p(x|\theta)}[log(p(x|\theta'))] \tag{22}$$

$$\nabla_{\theta'}\text{KL}(p(x|\theta)||p(x|\theta')) = \mathbb{E}_{p(x|\theta)}[\nabla_{\theta'}log(p(x|\theta))] - \mathbb{E}_{p(x|\theta)}[\nabla_{\theta'}log(p(x|\theta'))]$$

$$= 0 - \mathbb{E}_{p(x|\theta)}[\nabla_{\theta'}log(p(x|\theta'))]$$

$$= -\mathbb{E}_{p(x|\theta)}[\nabla_{\theta'}log(p(x|\theta'))] \tag{23}$$

$$\nabla_{\theta'}^2\text{KL}(p(x|\theta)||p(x|\theta')) = -\mathbb{E}_{p(x|\theta)}[\nabla_{\theta'}^2 log(p(x|\theta'))]$$

$$= -\mathbb{E}_{p(x|\theta)}[\mathbf{H}(l(\theta'|x))] \tag{24}$$

Evaluating at $\theta' = \theta$, then

$$\nabla_{\theta'}^2\text{KL}(p(x|\theta)||p(x|\theta'))|_{\theta'=\theta} = -\mathbb{E}_{p(x|\theta)}[\mathbf{H}(l(\theta|x))] = \mathbf{F}(\theta) \tag{25}$$

A.4 NGD update equation using Woodbury Identity

Assuming W_l and g_l to be vectors corresponding to the parameters of lth layer and their gradients,

NGD update
$$W_l(k+1) = W_l(k) - \alpha[F_l(W_l(k)) + \beta I]^{-1}g_l(k), \forall_{l=1,\ldots,L} \tag{26}$$

Woodbury Matrix Identity
$$(A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1} \tag{27}$$

Substituting $A = \beta I, C = I, U = \frac{J_l(k)^T}{m}$ and $V = J_l(k)$ in equation 27, we get

$$\left(\beta I + \frac{J_l(k)^T}{m}IJ_l(k)\right)^{-1} = \left(\beta I + \frac{J_l(k)^T J_l(k)}{m}\right)^{-1}$$

$$= (\beta I)^{-1} - (\beta I)^{-1}\frac{J_l(k)^T}{m}(I^{-1} + J_l(k)(\beta I)^{-1}\frac{J_l(k)^T}{m})^{-1}J_l(k)(\beta I)^{-1}$$

$$= \frac{I}{\beta} - \frac{I}{\beta}\frac{J_l(k)^T}{m}\left(I + J_l(k)\frac{I}{\beta}\frac{J_l(k)^T}{m}\right)^{-1}J_l(k)\frac{I}{\beta}$$

$$= \frac{I}{\beta} - \frac{J_l(k)^T}{m\beta}\left(I + \frac{J_l(k)J_l(k)^T}{m\beta}\right)^{-1}\frac{J_l(k)}{\beta}$$

$$= \frac{I}{\beta} - \frac{J_l(k)^T}{m\beta}\beta\left(\beta I + \frac{J_l(k)J_l(k)^T}{m}\right)^{-1}\frac{J_l(k)}{\beta}$$

$$= \frac{I}{\beta} - \frac{J_l(k)^T}{m}\left(\beta I + \frac{J_l(k)J_l(k)^T}{m}\right)^{-1}\frac{J_l(k)}{\beta}$$

$$\left(\frac{J_l(k)^T J_l(k)}{m} + \beta I\right)^{-1} = \frac{1}{\beta}\left(I - \frac{J_l(k)^T}{m}\left(\beta I + \frac{J_l(k)J_l(k)^T}{m}\right)^{-1}J_l(k)\right)$$

$$\therefore W_l(k+1) = W_l(k) - \alpha\frac{1}{\beta}\left(I - \frac{J_l(k)^T}{m}\left(\beta I + \frac{J_l(k)J_l(k)^T}{m}\right)^{-1}J_l(k)\right)g_l(k)$$

$$= W_l(k) - \frac{\alpha}{\beta}\left(g_l(k) - \frac{J_l(k)^T}{m}\left(\beta I + \frac{J_l(k)J_l(k)^T}{m}\right)^{-1}J_l(k)g_l(k)\right) \tag{28}$$

A.5 TENGraD Soori et al. [2021]

Notation:
$\odot$: Hadamard product between two matrices i.e. elementwise product

$\otimes$: Kronecker product

$||.||_2$: l_2 norm or euclidean norm

$*$: column-wise Khatri-Rao product

$M : i$: i^{th} column of matrix M

A deep neural network with L layers is a functional approximator in the form of $f(x, W)$ where x is the input and W are the parameters of the network. Let n be the total number of data samples and m be the batch size that is used to train the model. Then, for each layer l, let the input size be d_i^l and the output size be d_o^l. The weight of the layer l is denoted by $W_l \in \mathbb{R}^{d_i^l \times d_o^l}$, the input by $I_l \in \mathbb{R}^{d_i^l \times m}$ and the preactivation output by $O_l \in \mathbb{R}^{d_o \times m}$.

$$O_l = W_l^T I_l \tag{29}$$

With the sample (x, y) of size n, the model is trained to learn the parameters of the model that explain the data. This is usually done by minimizing an average loss function on the training data with respect to the model's parameters (equation 30). Usually, x is referred to as features and y as labels.

$$\mathcal{L}(W) = \frac{1}{n} \sum_{i=1}^{n} l(f(x_i, W), y) \tag{30}$$

Here, $W = [vec(W_1)^T, vec(W_2)^T, ..., vec(W_L)^T]$ denotes the vectorized form of all the parameters of the model. let p represents the total count of parameters i.e $W \in \mathbb{R}^p$.

Gradient and Jacobian: The gradient g of the loss function (equation 30) with respect to W is calculated over the batch data and often used for the training. This is the partial derivative of the $\mathcal{L}$ with W. Similarly, let g_l denote the gradient corresponding to the lth layer.

$$g = \nabla_W \mathcal{L}(W) = \frac{1}{m} \sum_{i=1}^{m} \nabla_W l(f(x_i, W), y_i) \in \mathbb{R}^p \tag{31}$$

$$\tag{32}$$

Backpropagation is used while training the neural networks during which each layer receives the partial derivatives of the loss with respect to its outputs for each data in the batch. Let's denote this derivative of the loss with respect to the preactivation output by $G_l \in \mathbb{R}^{d_o^l \times m}$. $G_l[i, j]$ denotes the partial derivative of loss with respect to i^{th} preactivation output of layer l for j^{th} data in the batch. Then, the gradient of the loss with respect to the parameters of the layer (g_l) is given by

$$g_l = \frac{1}{m} I_l G_l^T \tag{33}$$

For each layer l, let J_l denote the Jacobian of the loss with respect to each layer parameter for m data in a batch. These jacobians can be concatenated to form the Jacobian J with respect to all the parameters of the model.

$$J_l = (I_l * G_l)^T \in \mathbb{R}^{m \times d_i^l d_o^l} \tag{34}$$

$$J = [J_1, J_2, ..., J_L] \in \mathbb{R}^{m \times p} \tag{35}$$

A.5.1 Parameter Updates

Here, we will see how the updates in TENGraD is derived from the original NGD for efficiency and scalability purpose.

Original NGD Update at iteration k for DNN:

$$vec(W(k+1)) = vec(W(k)) - \alpha[F(W(k)) + \beta I]^{-1} vec(g(k)) \tag{36}$$

where α is the learning rate, F is the FIM. A non-negative damping factor β is added to prevent the singular case of F when the model is overparameterized to ensure the inversion of the matrix

in equation 36. The FIM coincides with the Gauss-Newton matrix when the output conditional distribution is assumed to be an exponential distribution.

$$
\begin{aligned}
F(W(k)) &= \mathbb{E}_{p(x,y)}[\nabla_W logp_W(y|x)\nabla_W logp_W(y|x)^T] \\
&\approx \frac{1}{m}\sum_{i=1}^{m}\nabla_W l(f(x_i, W(k), y_i)\nabla_W l(f(x_i, W(k), y_i)^T \quad \text{(Using Empirical FIM)} \\
&= \frac{1}{m}\sum_{i=1}^{m}\nabla_W l_i \nabla_W l_i^T \\
&= \frac{1}{m}J(k)^T J(k)
\end{aligned}
\tag{37}
$$

NGD Update with Block diagonal FIM: When the FIM is approximated using a block diagonal matrix, then each of layer can be updated separately using FIM corresponding to each layer (F_l). The update for layer l becomes

$$
F_l(W_l(k)) = \frac{1}{m}J_l(k)^T J_l(k)
\tag{38}
$$

$$
W_l(k+1) = W_l(k) - \alpha[F_l(W_l(k)) + \beta I]^{-1}g_l(k), \forall_{l=1,\ldots,L}
\tag{39}
$$

Time Efficient Exact FIM inversion: The inversion is the most computationally intensive operation in the equation 39. TENGrad computes the exact FIM inverse matrix for all the layers by using factorization and reusing intermediate values in the computation. Using Woodbury identity for each block, update changes into equation 40(proof in Appendix A.4).

$$
vec(W_l(k+1)) = vec(W_l(k)) - \frac{\alpha}{\beta}\left(vec(g_l(k)) - \underbrace{\frac{J_l(k)^T}{m}}_{C}\underbrace{\left(\frac{J_l(k)J_l(k)^T}{m} + \beta I\right)^{-1}}_{A}\underbrace{J_l(k)vec(g_l(k))}_{B}\right)
\tag{40}
$$

All the components in the equation 40 are computed efficiently in TENGraD. The gradient of layer l from the data i is given by $g_{l,i} = I_{l,:i}G_{l,:i}^T$(equation 33) which is a rank one matrix obtained by outer product. Then $[J_l J_l^T]_{i,j} = <vec(g_{l,i}), vec(g_{l,j})>$. We know that for the outer product $vec(uv^T) = u \otimes v$ which makes $[J_l J_l^T]_{i,j} = <I_{l,:i} \otimes G_{l,:i}, I_{l,:j} \otimes G_{l,:j}>$. Using $<u_1 \otimes v_1, u_2 \otimes v_2> = u_1^T u_2.v_1^T v_2$, we get $[J_l J_l^T]_{i,j} = I_{l,:i}^T I_{l,:j}.G_{l,:i}G_{l,:j}$. Extending this to the batch of size m, we have

$$
J_l J_l^T = (I_l * G_l)^T (I_l * G_l) = I_l^T I_l \odot G_l^T G_l = C_1 \odot C_2
\tag{41}
$$

The gram Jacobian is the Hadamard product of two small matrices of size $m \times m$. It can be efficiently computed with matrix product and without having to compute the jacobian. Part A in equation 40 involves small matrices and can be computed efficiently. Part B and C involve the use of jacobian and requires m times the parameter size memory requirement to store it. TENGraD circumvents this problem by storing small data structures and recomputing the jacobian as per necessity. Using equation 34 and properties of column-wise Khatri-Rao product, the C part in equation 40 can be computed as

$$
J_l vec(g_k) = (I_l * G_l)^T vec(g_l(k)) = ((g(k)^T I) \odot G)1
\tag{42}
$$

This involves two efficient matrix operations without having to form the jacobian. Then, the product of A and B term results in a vector $v \in \mathbb{R}^m$ requiring $J_l(K)^T.v$ as the final operation which can also be done efficiently.

$$
J_l^T = (I * G)v = I(v1^T \odot G^T)
\tag{43}
$$

With the use of the step in equation 41 for computing part A, equation 42 for computing part B, and equation 43 for part C, TENGraD computes the updates in equation 40 very efficiently in terms of time and space requirements.