Natural Gradient Descent for Neural Networks with Orthogonal Greedy Approximation

Luowei Yin

March 22, 2024

1 Natural gradient

Consider the PINN loss for a Poisson PDE:

$$\mathcal{L}(u_\theta) = \frac{1}{2}\| - \Delta u_\theta - f\|^2_{L^2(\Omega)} + \frac{\tau}{2}\|u_\theta - g\|^2_{L^2(\partial\Omega)}.$$

One may try to obtain analytically $\nabla_u \mathcal{L}(u^k)$ and approximate a step of gradient descent by neural networks. However this is not feasible because the gradient does not exist in $L^2(\Omega)$. Updating u by this gradient will make its regularity insufficient for the state PDE to be solved. If one try to project the gradient by smoothing operator (Ritz map, or Hilbertian approach) $H^{-1} \to H^1$, this amounts to solving another PDE.

Another technique to approximate this function gradient is to use the natural gradients, whose main idea is to project the function gradient into the tangent space of the model manifold.

Define the manifold $\mathcal{M} = \{u_\theta : \theta \in \mathbb{R}^p\}$, and the generalized tangent space at θ by $T_{u_\theta}\mathcal{M} := \text{span}\{\frac{\partial u_\theta}{\partial \theta_i} : i = 1 \cdots p\}$. For a infinitesimal move of θ, it generates a move of u_θ along the manifold.

Definition 1. *Consider an infinitesimal step $\frac{d\theta}{dt} = h$ for some $h \in \mathbb{R}^p$. We have*

$$\frac{du_\theta}{dt} = \sum_{i=1}^p \frac{\partial u_\theta}{\partial \theta_i}\frac{d\theta_i}{dt} = \sum_{i=1}^p \frac{\partial u_\theta}{\partial \theta_i} h_i.$$

This evolving direction of u_θ is called the push-forward at u_θ by h, denoted by $\#h$.

The tangent space is a finite dimensional linear space, consisting of all possible push-forward at u_θ. Note that by the definition of tangent space, it is always a subset of the space where u_θ lies in. The natural gradient aims to find the best approximation of $\nabla_u \mathcal{L}$ in $T_{u_\theta}\mathcal{M}$, which is given by projection:

$$\tilde{\nabla_u}\mathcal{L} = \arg\min_{v \in T_{u_\theta}\mathcal{M}} \|v - \nabla_u \mathcal{L}\|^2_{L^2(\Omega)} = \Pi_{T_{u_\theta}\mathcal{M}}(\nabla_u \mathcal{L}).$$

To find the approximated gradient, it suffices to solve the cofficients h_i with respect to some metric. More specifically, taking the canonical L^2 inner product as metric,

$$\begin{aligned} h^* &= \arg\min_{h \in \mathcal{R}^p} \|\sum_{i=1}^p \frac{\partial u_\theta}{\partial \theta_i} h_i - \nabla_u \mathcal{L}\|^2_{L^2(\Omega)}, \\ \#h^* &= \tilde{\nabla}_u \mathcal{L}. \end{aligned}$$

If the function gradient $\nabla_u \mathcal{L}$ is not explicitly known or behaves badly such that integrals are diffucult to evaluate, one can apply the following workaround.

Assuming that $T_{u_\theta}\mathcal{M}$ lies in a separable and uniformly convex Hilbert space. Then there is an orthogonal split for $\nabla_u \mathcal{L}$:

$$\nabla_u \mathcal{L} = \tilde{\nabla}_u \mathcal{L} + \nabla_u \mathcal{L}^\perp.$$

Where $\tilde{\nabla}_u \mathcal{L}$ is the projcetion part and the orthogonal part $\nabla_u \mathcal{L}^\perp$ satisfy $(\nabla_u \mathcal{L}^\perp, \frac{\partial u_\theta}{\partial \theta i})_{L^2(\Omega)} = 0$ for each $i = 1 \cdots p$. Now we can test $\nabla_u \mathcal{L}$ by each basis function $\frac{\partial u_\theta}{\partial \theta j}$, yielding

$$(\nabla_u \mathcal{L}, \frac{\partial u_\theta}{\partial \theta j})_{L^2(\Omega)} = (\tilde{\nabla}_u \mathcal{L}, \frac{\partial u_\theta}{\partial \theta j})_{L^2(\Omega)} = \sum_{i=1}^{p} (\frac{\partial u_\theta}{\partial \theta i}, \frac{\partial u_\theta}{\partial \theta j})_{L^2(\Omega)} h_i^*.$$

An obvious benefit of this formulation comes from $(\nabla_u \mathcal{L}, \frac{\partial u_\theta}{\partial \theta j})_{L^2(\Omega)} = \frac{\partial \mathcal{L}}{\partial \theta_j}$ by chain rule. If u_θ is a neural network, this gradient $\frac{\partial \mathcal{L}}{\partial \theta_j}$ can be easily and precisely obtained by auto-differentiation. The burden of evaluating integrals involving $\nabla_u \mathcal{L}$ is transferred to the Gram matrix. For neural networks, u_θ is usually smooth, hence integrals in Gram matrix are relatively easier to evaluate. Now it suffices to solve $h^* \in \mathbb{R}^p$ such that

$$M h^* = \nabla_\theta \mathcal{L}.$$

where M is the Gram matrix. This is the most common formulation of natural gradients. But numerically the Gram matrix usually becomes singuler during evolution. (is accuracy of h^* really important here?).

In natural gradient descent, solving the gradient descent direction is equivalent to finding the best approximation of function gradient in the tangent space of model manifold with respect to a given metric. The Gram matrix becoming singular implies highly correlated tangent vectors. While, this correlation does not imply reduced expressivity. Several problem are raised here: how to characterize the redundancy of tangent vectors, and is it possible to find a subset of tangent vectors that generates nearly best approximation of function gradient? When the training gets stuck at a local minima, it means the function gradient becomes orthogonal to the tangent space. The natural gradient displays another possibility that the tangent vectors are heavily "clustered" and becomes non informative.

1.1 Orthogonal Greedy Algorithm

The Gram matrix becoming singular implies the increasing correlation of tangent vectors, and a degradation of tangent space. When tangent vectors are highly correlated, the projection error onto tangent space becomes large, hence gradient descent becomes less effective. Hence the following method is motivated to reduce the redundancy of tangent vectors by choosing only several, instead of all basis vectors that achieves a good approximation of target gradient.

Selecting k tangent vectors that minimize a projection error of $\nabla_u \mathcal{L}$ is generally infeasible, since it requires to go through all the combinations. We may instead try to do this in a sequential greedy manner. Suppose each gradient descent step only one parameter is to be updated, the optimal choice of tangent vector is trivially given by:

$$i_1 = \arg\min_{i=1\cdots p} \|\nabla_u \mathcal{L} - \frac{(\nabla_u \mathcal{L}, \frac{\partial u_\theta}{\partial \theta i})_{L^2(\Omega)}}{\|\frac{\partial u_\theta}{\partial \theta i}\|^2} \frac{\partial u_\theta}{\partial \theta i}\|^2 \tag{1}$$

$$= \arg\max_{i=1\cdots p} \frac{|\partial_{\theta_i} \mathcal{L}|}{\|\frac{\partial u_\theta}{\partial \theta i}\|} \tag{2}$$

This produces a residual defined as $r_1 = \nabla_u \mathcal{L} - \Pi_{H_1}(\nabla_u \mathcal{L})$, where $H_1 = \text{span}(\partial_{\theta_{i_1}} u_\theta)$. Continuing this approach, the second vector best approxiamtes the residual r_1:

$$i_2 = \arg\min_{i=1\cdots p} \|r_1 - \frac{(r_1, \frac{\partial u_\theta}{\partial \theta i})_{L^2(\Omega)}}{\|\frac{\partial u_\theta}{\partial \theta i}\|^2} \frac{\partial u_\theta}{\partial \theta i}\|^2 \tag{3}$$

Hence the algorithm is summarized as follows:

Denote: Index set of selected basis I_k, linear space spanned by index set $H_k := \text{span}\{\partial_{\theta_i} u_\theta, i \in I_k\}$. Residual $r_k := \nabla_u \mathcal{L} - \Pi_{H_k}(\nabla_u \mathcal{L})$, and the coefficient vector h_k of $\Pi_{H_k}(\nabla_u \mathcal{L})$ in the tangent space. The one-dimensional subspace spanned by each $\partial_{\theta_i} u_\theta$ is denoted by S_i. Initialize $I_0 = \emptyset$, the $(k+1)$-th

new index to be added into I is given by:

$$\begin{aligned} i_{k+1} &= \arg\min_{i\notin I_k} \|r_k - \Pi_{S_i}(r_k)\|^2 \\ &= \arg\min_{i\notin I_k} \|r_k - \frac{(r_k, \frac{\partial u_\theta}{\partial \theta i})_{L^2(\Omega)}}{\|\frac{\partial u_\theta}{\partial \theta i}\|^2}\frac{\partial u_\theta}{\partial \theta i}\|^2 \\ &= \arg\min_{i\notin I_k} \frac{(r_k, \frac{\partial u_\theta}{\partial \theta i})^2}{\|\frac{\partial u_\theta}{\partial \theta i}\|^2} - 2\frac{(r_k, \frac{\partial u_\theta}{\partial \theta i})^2}{\|\frac{\partial u_\theta}{\partial \theta i}\|^2} \\ &= \arg\max_{i\notin I_k} \frac{|(r_k, \frac{\partial u_\theta}{\partial \theta i})|}{\|\frac{\partial u_\theta}{\partial \theta i}\|} \end{aligned}$$

Since $r_k = \nabla_u \mathcal{L} - \Pi_{H_k}(\nabla_u \mathcal{L}) = \nabla_u \mathcal{L} - (G^{-1}_{[I_k]}(\nabla_\theta \mathcal{L})_{k,[I_k]})^T[\partial_{\theta_i} u_\theta, i \in I_k]$, we have

$$\begin{aligned} i_{k+1} &= \arg\max_{i\notin I_k} \frac{|(r_k, \frac{\partial u_\theta}{\partial \theta i})|}{\|\frac{\partial u_\theta}{\partial \theta i}\|} \\ &= \arg\max_{i\notin I_k} \frac{|(\nabla_u \mathcal{L} - (G^{-1}_{[I_k]}(\nabla_\theta \mathcal{L})_{k,[I_k]})^T[\partial_{\theta_i} u_\theta, i \in I_k], \frac{\partial u_\theta}{\partial \theta i})|}{\|\frac{\partial u_\theta}{\partial \theta i}\|} \\ &= \arg\max_{i\notin I_k} \frac{|(\nabla_u \mathcal{L}, \frac{\partial u_\theta}{\partial \theta i}) - ((G^{-1}_{[I_k]}(\nabla_\theta \mathcal{L})_{k,[I_k]})^T[\partial_{\theta_i} u_\theta, i \in I_k], \frac{\partial u_\theta}{\partial \theta i})|}{\|\frac{\partial u_\theta}{\partial \theta i}\|} \\ &= \arg\max_{i\notin I_k} \frac{|\partial_{\theta_i} \mathcal{L} - ((G^{-1}_{[I_k]}(\nabla_\theta \mathcal{L})_{k,[I_k]})^T[\partial_{\theta_i} u_\theta, i \in I_k], \frac{\partial u_\theta}{\partial \theta i})|}{\|\frac{\partial u_\theta}{\partial \theta i}\|} \\ &= \arg\max_{i\notin I_k} \frac{|\partial_{\theta_i} \mathcal{L} - (Gh_k)_{[i]}|}{\|\frac{\partial u_\theta}{\partial \theta i}\|} \\ &= \arg\max_{i\notin I_k} \frac{|\nabla_\theta \mathcal{L} - Gh_k|}{[\|\frac{\partial u_\theta}{\partial \theta i}\|, i = 1\cdots p]}. \end{aligned}$$

where the division in last line is element-wise. Note that all matrix and vectors are pre-computed, hence no integration is needed during iteration. Once a new index is obtained, we update:

$$I_{k+1} = I_k \cap \{i_{k+1}\}, h_{k+1,[I_{k+1}]} = G^{-1}_{[I_{k+1}]} \nabla_\theta \mathcal{L}_{[I_{k+1}]}, h_{k+1,[I^c_{k+1}]} = 0.$$

Iterating over $k = 1\cdots K$, we obtain a sequence of projection that converges to the best approximation of $\nabla_u \mathcal{L}$ in the tangent space. Moreover, the residual is also non-increasing in norm, ans can be estimated by:

$$\|r_k\|^2 = (Gh_k - 2\nabla_\theta \mathcal{L})^T h_k + C$$

where the constant $C = \|\nabla_u \mathcal{L}\|^2$ is fixed during iteration.

2 Numerical evidence

2.1 Testing on least squares

We test the performance of NGD on least squares problem:

$$\min_{\theta\in\mathbb{R}^p} \frac{1}{2}\|u_\theta - g\|^2_{L^2(\Omega)}.$$

where the data $g = \sin(\pi x)\sin(\pi y) + \sin(\frac{3}{2}\pi x)\sin(\frac{3}{2}\pi y)$, domain $\Omega = [0,1]^2$, and the integral is evaluated by Monte Carlo.

Test on shallow networks

Layers: $[2, 30, 1]$ with tanh activation function. Target function:

$$g = \sin(\pi x)\sin(\pi y) + \sin(\frac{3}{2}\pi x)\sin(\frac{3}{2}\pi y)$$

Logarithmic grid search in interval $[0, 1]$ with 30 grids. Greedy step is 10. Parameter gradient will not be cut. Preconditioned gradient is the minimum-norm solution.

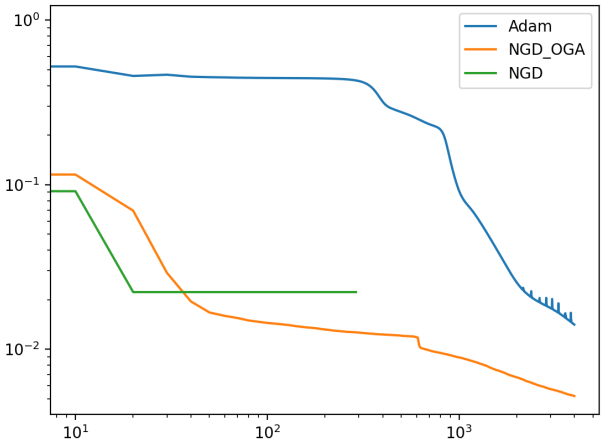

Figure 1: Least sqaures by shallow NN.

Test on multilayer perceptron

Layers: $[2, 20, 20, 1]$ with tanh activation. Greedy step is 20. Parameter gradient will not be cut. Preconditioned gradient is the minimum-norm solution.

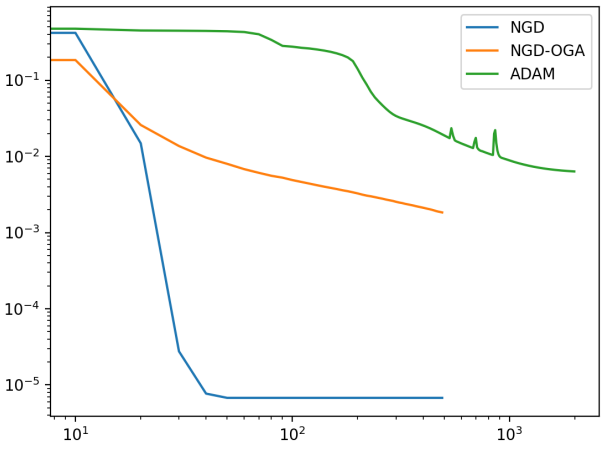

Figure 2: Least sqaures by multilayer NN.

2.2 Testing on Poisson PDE

We test the performance of NGD on 2D Poisson PDE on squared domain, with ground truth $u^* = \sin(\pi x)\sin(\pi y)$. As reported on their paper, the E-NGD sometimes get stuck in local optima and the loss value remains large. There are two possible reasons: 1. in the very first few steps, due to insuitable initialization, the Gram matrix tend to be singular and it yields an astronomically large gradient scale. One can remedy by cutting gradient into a reasonable range, or pre-training by stanrdard gradient descent for a few steps, as commented on paper. 2. Sometimes the Gram matrix is always too singular and in each step the gradient is always large in scale. Cutting gradient does not give a valid descent direction, and not cutting gradient leads to a bad step. Using backtracking line search instead of grid search might help.

Test on Poisson example via E-NGD

Firstly we compare E-NGD and E-NGD-OGA. The boundary penalty weight is 100, and architecture is layer $[2, 20, 20, 1]$ with tanh activation. To prevent exploding parameter, we cut each parameter gradient into $[-1000, 1000]$. Logarithmic grid search is applied to find suitable stepsize.

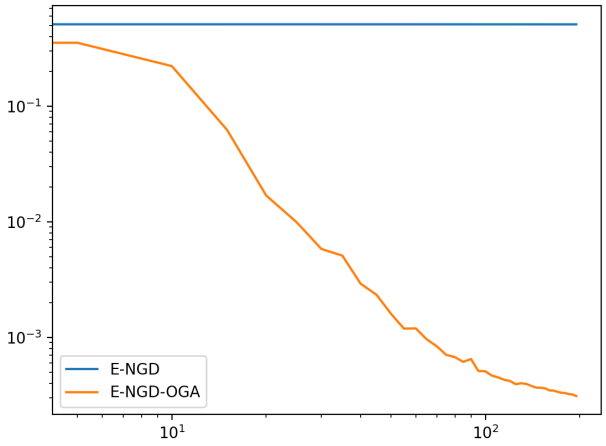

Figure 3: Poisson test 1.