CS598: Deep Generative & Dyn. Models
Final Report: Not All Sensors Are Equal
Optimal Sensor Points for Operator Learning

Nicolas Nytko, Sean Farhat, Nathanael Assefa, (Alexey Voronin)

July 10, 2023

Abstract

There has been a wealth of research in numerical solvers for partial differential equations dedicated to the optimization of polygonal meshes used to approximate solutions. The careful construction of such meshes can affect the overall quality of the obtained approximation; however, little has been conducted in the analogous domain of operator learning for PDEs. This study introduces a method for optimizing the placement of sensor points for DeepONets by means of a nested hyperparameter search. A comparison ablation study is conducted for the steady-state diffusion and time-dependent wave equations, showing promising results indicating that sensor points tailored towards a specific PDE class can improve the resulting solution quality.

1 Introduction

Partial Differential Equations (PDEs) provide a mathematical framework for describing and understanding many physical phenomena in science and engineering, from the behavior of fluids and materials to the dynamics of electric and magnetic fields. The solution of PDEs is essential for making predictions, designing and optimizing systems, and developing new technologies. Solving PDEs analytically can be challenging, and in many cases, impossible. Therefore, numerical methods are often used to approximate the solutions of PDEs.

To solve numerical PDEs using traditional numerical methods, one must turn the continuous problem into something discrete that is solvable on a computer: this is commonly done by the finite element method (FEM), where a problem domain is approximated by a discrete triangulation (mesh) upon which the PDE is satisfied discretely (think Taylor approximation). Once the PDE has been discretized and approximated, the resulting system of equations is solved numerically such that the solution minimizes the residual on the mesh. This mesh ultimately determines how fast we can solve the system, and also to what accuracy we are guaranteed a solution. Thus our intuition motivates that the selection of discretization points is likely also important in an operator learning setting.

DeepONets [8] are a relatively recent framework for learning nonlinear operators between regular function spaces that operate in a *mesh-free* way. Input parameters, such as forcing terms or diffusivity fields, are evaluated at some fixed set of *sensor points*, then passed into the network which allows evaluation at any arbitrary set of *evaluation points*. These sensor points are fixed, however, for the entire dataset that the network is trained on. In most ONet literature, these sensor points are constrained to a uniform grid over the problem domain. However, this raises the question on whether there is some more optimal set of sensor points that would result in a more accurate learned operator evaluation.

2 Random Walk Investigation and Analysis

Our original project approach considered mapping the unstructured grid to a structured one using a generative process to learn the backward perturbations of an unstructured mesh into a structured mesh via a

Diffusion model. Full details for this are given in Appendix A.

We decided to pivot away from this approach because although the process is feasible in theory, learning a meaningful backward process could be prohibitive both in complexity and in analysis. The proposed pointwise noising for optimal exploration may require a prohibitively large de-noising network and the proposed rejection sampling can skew the distribution of the accepted Gaussians and invalidate part of the analysis in [3] for our model.

3 Related Work

The related works for our original idea are somewhat sparse (pun intended). There is some old work by Douglas et. al [1] to transform problems to uniform meshes, however the process of determining a mapping between the domains is done in a largely manual way. Additionally, this has been developed, it seems, for nonuniform structured problems. We were unable to find any related works that feature learning in any significant way. Thus, we hope that this project will have some novel impact.

Our final direction concerned Neural Operators [5, 6, 8]. While the area is quite new, there is not much work investigating optimization of the learning process itself, but rather foundational results such as universal expressivity [4, 8]. Our work seeks to extend the framework of DeepONets [8] by considering the value of learning domain-specific hyperparameters.

4 Methods

An ONet can be described as a function G that takes in a function u and outputs a function $G(u)$. We cannot possibly use the infinite-dimensional representation of every function, so we have to discretize both the input and output. ONets allow us to probe the output function at any points $\mathbf{y}$, as those are plugged into the trunk portion of the network. However, the input function u must be sampled at a set of predetermined sensor points $\mathbf{x} = x_1, \ldots, x_m$. It is important to note that these sensors are chosen *beforehand*, and while we can choose as many as we would like, once we have chosen them, *all* input functions must be sampled at these sensors. So an ONet takes in $u(\mathbf{x})$ into the branch and $\mathbf{y}$ into the trunk (Figure 1).

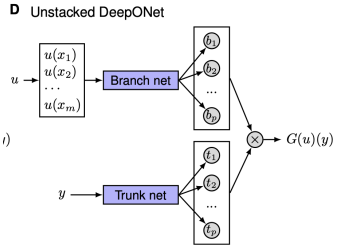


Figure 1: The ONet Architecture. The branch takes in the input function u evaluated at the fixed sensors $x_1, \ldots, x_m$, while the trunk takes in the sensors y for the output function $G(u)$.

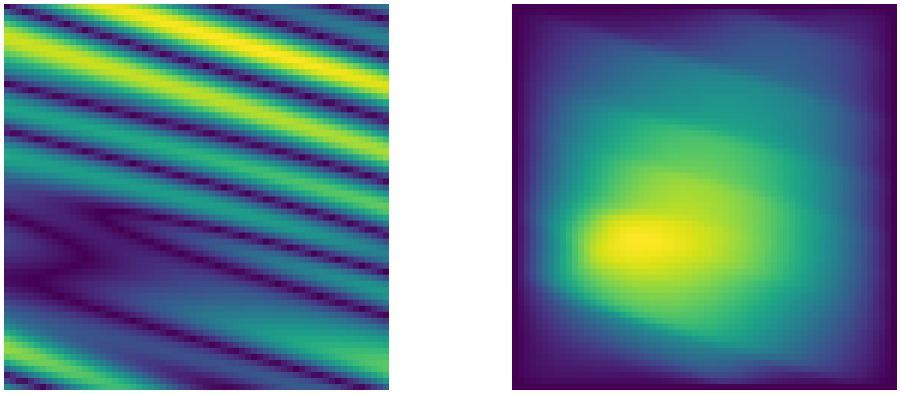
While this scheme certainly works in practice, we believe that **not all sensors are equal**; in fact, certain problems are better off if the sensors are placed elsewhere. Therefore, we propose to optimize the sensors as well as the ONet itself by treating the sensors as hyperparameters in the algorithm.

Early hyperparameter optimization literature revolved around brute force approaches such as grid search or Bayesian optimization. More recently, meta-learning has made a strong case for jointly optimizing the model and a set of hyperparameters. The general algorithm corresponds to allowing the model, parameterized by θ, to learn for a few steps on the training set T, then validating (V) the model and optimizing the hyperparameters λ to minimize the outer loss,

$$\arg\min_{\lambda} \mathcal{L}_V \left(\lambda, \theta - \eta \nabla_\theta \mathcal{L}_T(\lambda, \theta)\right). \tag{1}$$

The equation above considers the inner optimization to be one step of gradient descent, though WLOG it can be any number of steps with any optimizer. In the best case, the inner optimization will converge to the optimal θ^*, however this isn't guaranteed. We find that this is not necessary in order for learning to progress. Thus, our learning problem is

$$\arg\min_{\mathbf{x}} \mathcal{L}_V \left(u(\mathbf{x}), y, \arg\min_{\theta} \mathcal{L}_T(u(\mathbf{x}), y, \theta)\right). \tag{2}$$



(a) A randomly generated function of $\kappa(x, y)$ evaluated over the domain.

(b) The numerically computed solution for the given $\kappa(x, y)$.

Figure 2: An example data point in the generated set of data for the diffusion equation.

It is naturally of interest to allow the inner optimization to run for more iterations such that the model used to optimize $\mathbf{x}$ is well-trained. However, this corresponds to unrolling the gradients in the optimization, which incurs a high computational cost proportional to the number of inner steps, as well as the possibility of suffering from problems such as vanishing gradients. A discussion of how we approached this problem can be found in Appendix B.

A question of generalizability must also be addressed. While it makes sense for grid points to be tuned for different problems, ONets have the power to deal with all inputs functions (in theory), and one sensor grid is not universally supreme. Considering this, we assume the setting where the network is trained on a class of related functions so that the learned sensors will be useful to that family of problems.

5 Data

We generate example data for two sample families of PDEs, the steady-state diffusion PDE and unsteady wave equation on the unit square. The implementation of these is detailed in the remainder of this section.

5.1 Steady-State Diffusion

The steady-state diffusion PDE is an elliptic partial differential equation describing the long-term behavior of microscopic particles suspended in some solution. Formally, we are solving

$$\nabla \cdot (\kappa(x, y)\nabla u) = 1, \tag{3}$$

on the domain $\Omega = [0, 1]^2$, with a homogeneous Dirichlet boundary condition $u(\partial\Omega) = 0$. Here, $\kappa(x, y)$ is the diffusivity of the material at some point and $u(x, y)$ is the unknown density we would like to solve for. We randomly generate $\kappa(x, y)$ using simplex noise [2]. Simplex noise, implemented in OpenSimplex, is a gradient noise function $\sigma : \mathbb{R}^2 \to \mathbb{R}$ that produces smooth isotopic noise. We use σ to randomly generate the κ like

$$\kappa(x, y; \theta, s_1, s_2) := \sigma\left(\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} s_1 & \\ & s_2 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}^T \right), \tag{4}$$

with $s_1, s_2 \sim \mathcal{N}(0, 10); \theta \sim \mathcal{U}(0, 2\pi)$. So, a random linear transformation to the input of the simplex function.

These are discretized onto a structured 64×64 mesh using the Firedrake library [9, 10] on Equation 3. We save the data in tuples of the form

$$\mathcal{D} = \left\{ u|_{x,y}, \ \ \kappa|_{x,y} \right\}, \tag{5}$$

where $u|_{x,y}$ denotes the numerical solution $u(x, y)$ evaluated at the regular 64×64 grid points x, y (and similarly for κ). We generate a total of 1,000 samples for a training set, and 300 samples for a validation set. See Figure 2 for an example data point that was generated.

5.2 Time-Dependent Wave Equation

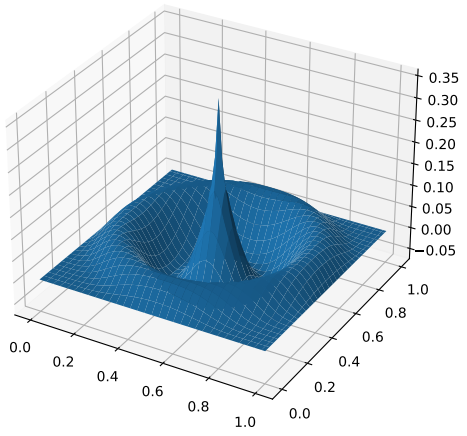


Figure 3: An example simulation of the wave equation with a unit perturbation applied to the center of the domain.

The wave equation is a linear partial differential equation of the form

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u. \tag{6}$$

We again solve this on the domain $\Omega = [0, 1]^2$ with homogeneous Dirichlet boundary conditions $u(\partial\Omega) = 0$. These specific boundary conditions allow the waves to *reflect* when they hit the boundary of our domain, allowing for more interesting behavior. This is discretized in space using centered finite differences; we obtain the standard 5-point stencil [11] which we represent by the matrix operator A. We also use centered finite differences in time, giving us the implicit recurrence

$$\frac{1}{\Delta t^2}\left(u^{t-1} - 2u^t + u^{t+1}\right) = c^2 A u^{t+1}, \tag{7}$$

which we can rewrite as the matrix equation

$$\left(I - c^2 \Delta t^2 A\right) u^{t+1} = 2u^t - u^{t-1}. \tag{8}$$

So, we have a linear solve to obtain the next time step based on the current and previous time steps. Luckily the linear system remains the same during all time steps, so we can reuse the matrix factorization. For this problem, we use $c = 0.2$ and $\Delta t = 0.01$. Like above, this is discretized on a regular 64×64 square of grid points.

We generate random samples by initializing the solution with a random perturbation applied to some point in the domain. This is allowed to simulate for several time steps, until the waves have propagated to much of the domain, at which point we capture the solution for a fixed number of time steps and save this to our dataset. Overall, we perform 25 "runs" for the training set, and 10 for the testing set. During each run, we capture 150 snapshots (a subset of what was actually computed), giving us 3750 training samples and 1500 testing samples. See Figure 3 for an example initialization of the PDE.

We save the dataset as a triple of "snapshots"; each triple describes sequential snapshots of the PDE obtained as a sliding window,

$$\mathcal{D} = \left\{u^{t-2}|_{x,y}, u^{t-1}|_{x,y}, u^t|_{x,y}\right\}. \tag{9}$$

The ONet is tasked with learning some time step of the PDE given the previous two time steps; giving it two previous time steps allows the network to extrapolate the momentum/velocity of the waves.

6 Training Results

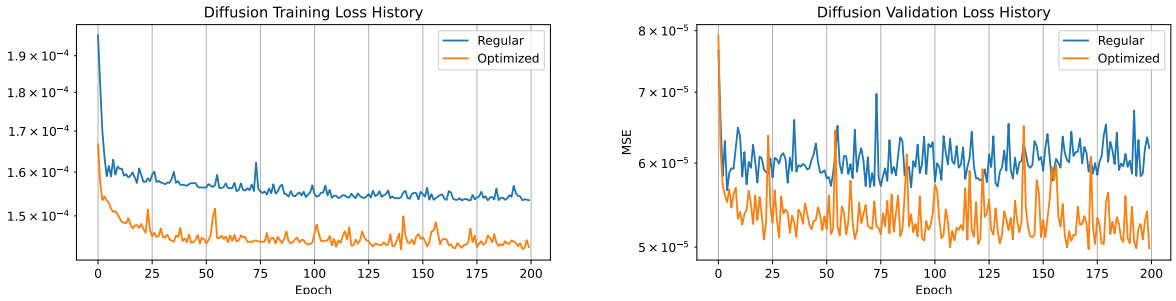
Overall, we see improvements to the trained ONet when we also optimize over the sensor points. As an ablation, we perform the same training routine with the same number of "outer" and "inner" optimizations, but turn off the actual optimization for the sensor points; thus this ONet is learning on the standard uniform grid of sensors. The parameters used for optimization are given in Table 1.

Problem	Outer Ep.	Inner Ep.	Outer Opt.	Inner Opt.	Inner LR	Outer LR
Diffusion	200	50	RMSProp	Adam	10^{-4}	10^{-3}
Wave	200	100	RMSProp	Adam	10^{-4}	10^{-3}

Table 1: Parameters used to train an ONet on both example PDEs.

Problem	Opt. Points	Training MSE	Testing MSE
Diffusion	No	1.53×10^{-4}	6.20×10^{-5}
	Yes	1.43×10^{-4}	4.99×10^{-5}
Wave	No	7.06×10^{-6}	7.36×10^{-6}
	Yes	2.02×10^{-6}	2.03×10^{-6}

Table 2: Training and testing mean squared errors when sensor points are optimized, vs when they are fixed to a uniform grid. Optimizing the sensor points leads to a decrease in the mean-squared-error.



(a) Train Loss

(b) Validation Loss

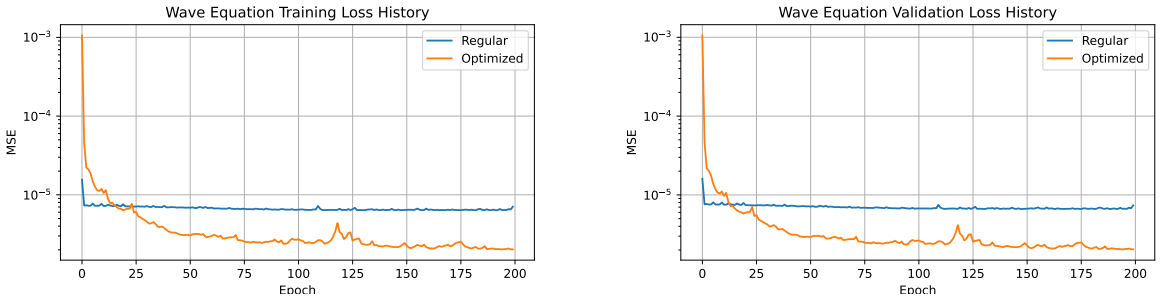
Figure 4: MSE Losses for Diffusion Problem

Allowing the optimizer to find better sensor points gives a noticeable decrease to the final MSE loss, as shown in Figures 4 and 5 and Table 2. In the case of the diffusion equation, the sensor points align themselves much closer to the boundary (Figure 6a), which matches our intuition of the physics of the problem; more resolution is needed at the boundary to represent our PDE solution. For the wave equation, the sensor points are much closer to the uniform grid with exception of those close to the boundary (Figure 6b). It is still interesting to note that such minor movement of the overall sensors can still decrease the overall MSE as compared to the ONet trained on the uniform points.

7 Conclusions

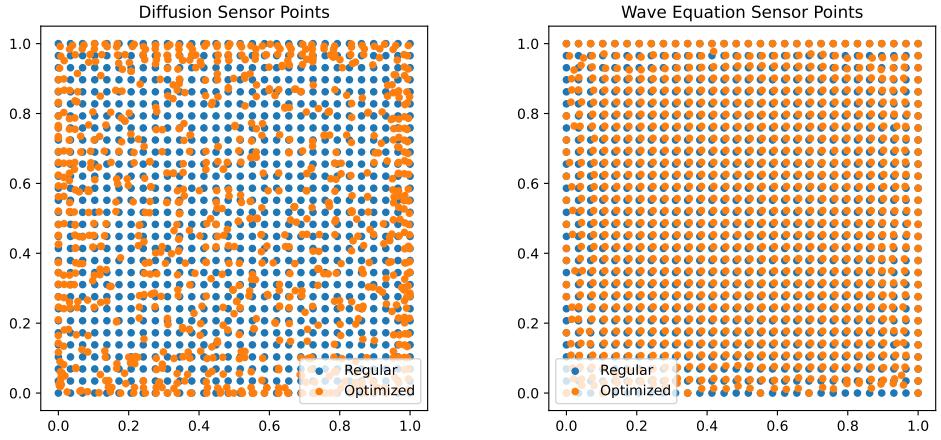
In this work, we have presented an algorithm for the simultaneous optimization of sensor points and network weights for a DeepONet network. We generated example data for two families of PDEs and trained DeepONets with, and without the sensor point optimization. The results imply that even for minor movement of the sensor points from the initial uniform grid, the overall loss is further minimized when the optimized sensor points are used.

Future work could investigate this idea into other families of problems as well. In addition, most off-the-shelf PDE solvers can take in a set of grid points for discretization, so it would be interesting to see if the



(a) Train Loss (b) Validation Loss

Figure 5: MSE Losses for Wave Problem



(a) Diffusion Sensors (b) Wave Equation Sensors

Figure 6: Sensor points optimized over both PDE datasets.

optimized set of sensor points would be allow a PDE solver to find a better solution. It is also worth looking into subjecting our method to the scrutiny of implicit bias in these formulations, as described in [12].

Analytical studies of point placement for PDEs offer a basis and operator-dependent intuitive explanation for the optimal placement of N points. The approach focuses on ensuring that the basis functions are point-wise integrated with minimal overlap and motivated by approaching the problem from a toy one-point quadrature approach and optimizing additional points based on the placement of prior points to produce a smoother basis approximation to the operator. An obvious limitation is that one-point quadrature is only possible for equations that have a fundamental solution or known Green's function. We're excited by the prospect of this approach producing useful and needed empirical guidance for sensor point placement in more difficult PDEs.

References

[1] C. C. Douglas, S. Malhotra, and M. H. Schultz. A characterization of mapping unstructured grids onto structured grids and using multigrid as a preconditioner. *BIT Numerical Mathematics*, 37(3):661–677, sep 1997.

[2] Stefan Gustavson. Simplex noise demystified. https://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf, 2005. Accessed 2023-03-07.

[3] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models, 2020.

[4] Nikola Kovachki, Samuel Lanthaler, and Siddhartha Mishra. On universal approximation and error bounds for fourier neural operators, 2021.

[5] Nikola Kovachki, Zongyi Li, Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Neural operator: Learning maps between function spaces, 2023.

[6] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations, 2021.

[7] Jonathan Lorraine, Paul Vicol, and David Duvenaud. Optimizing millions of hyperparameters by implicit differentiation, 2019.

[8] Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Em Karniadakis. Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 3(3):218–229, mar 2021.

[9] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly. Firedrake: Automating the finite element method by composing abstractions. *ACM Trans. Math. Softw.*, 43(3):24:1–24:27, December 2016.

[10] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. McRae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake: automating the finite element method by composing abstractions. *ACM Trans. Math. Softw.*, 43(3):24:1–24:27, 2016.

[11] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, 2003.

[12] Paul Vicol, Jonathan Lorraine, Fabian Pedregosa, David Duvenaud, and Roger Grosse. On implicit bias in overparameterized bilevel optimization, 2022.

A Original Diffusion Process

For an N point grid in 2 dimensions, the coordinates of the points belong in an $N \times 2$ matrix x in order to perturb the mesh in a natural fashion via a set of coordinates representing degrees of freedom.

Using the diffusion algorithm proposed by [3]. The forward process begins by sampling an x_0 coordinate matrix from a structured mesh and iteratively sampling from a time-dependent Gaussian$q(x^{(t)}|x^{(t-1)}) := \mathcal{N}\big(x^{(t)}; \sqrt{1-\beta_t}x^{(t)}, \beta_t\mathbb{I}\big)$. The variance schedule β_t is a time-dependent hyper-parameter and we can use the point-wise Gaussian property to product through the noise at once and generate the noised input at a desired time T producing $q(x^{(t)}|x^{(0)}) = \mathcal{N}\big(x^{(t)}; \sqrt{\tilde{\alpha}_t}x^{(0)}, (1-\tilde{\alpha}_t)\ \mathbb{I})\big)$ where $\alpha_t := 1-\beta_t, \tilde{\alpha}_t = \prod_1^t \alpha_i$. We can interpret this as taking each coordinate in a 2-D plane and perturbing it by a little noise. An important assumption we make is that the connections between the points does not change. This ensures that the A matrix at each step has the same sparsity pattern and will (for now) reduce the complexity and scope of the learning problem; we only need to modify nonzero entries in the linear operator when we perform the noising/de-noising process.

The reverse process is to start with a noisy set of coordinates which represent an unstructured mesh and iteratively remove the noise. This is achieved by taking a similar set of iterative steps by sampling Gaussians,

$$p\big(x^{(t-1)}|x^{(t)}\big) := \mathcal{N}\big(x^{(t)}; \mu_\theta(x^{(t)}, t), \Sigma_\theta(x^{(t)}, t)\big). \tag{10}$$

Unfortunately, this diffusing the coordinates is not sufficient. While we can learn to de-noise coordinates, this is not entirely useful by itself; ultimately the linear operator encodes some physical properties of the PDE itself and we wish to find some automated way to transfer the existing information to the structured mesh. Therefore, we will need to have a second diffusion process as well.

A.1 Coupled Diffusion Processes

We will also need to run a diffusion on the matrix A in parallel. Why? Providing just a set of noisy coordinates to be re-structured is not sufficient to solve any problem; we need to also provide the matrix that encodes important information about the underlying problem at hand. However, this is not straightforward.

the diffusion process on A must fit into the noising paradigm of DDPM to make the de-noising process work, but it must also be *consistent* with its corresponding $x^{(t)}$ at step t. To ensure these are both satisfied, we leverage the freedom given to us by the choice of variance schedules. We can choose the β's for the coordinate diffusion process normally, e.g. following the paper's implementation, but we can also carefully choose the variance schedule for the linear operator diffusion:

It is an established fact that for the finite element discretization for a 2D diffusion problem, the entries of the stiffness matrix A correspond to $\pm\kappa_{i,j}/h_{i,j}$ for some spatially dependent edge length $h_{i,j}$ and diffusivity $\kappa_{i,j}$. Ensuring that the perturbations of A will correspond to the coordinates correctly requires representing this profile in the scheduler coordinate α_t to obtain a mean $\sqrt{1-\beta_t}$ and variance $\beta_t\mathbb{I}$. We can select the scheduler for the restricted Sparse Gaussian applied to A at time T to be parameterized by the scheduler of the general form

$$\beta_{A_{ij}} = 1 - \frac{j\sqrt{1-\beta_j} + i\sqrt{1-\beta_i}}{A_{ij}} \text{ for every t.}$$

In the most general form, Here β_i is the variance schedule for the i^{th} coordinate and β_j is the variance schedule for the j^{th} coordinate while $\beta_{A_{ij}}$ is the variance schedule for A_{ij} at time T. This point-wise definition of the variance scheduler is a more general form of the tensor-specific selection for time t, which results from setting all the (i,j) indexed β values to be equal to one another. Since the Diffusion algorithm proposed by [3] employs linear operations on the noise at every time step T, the results generalize point-wise variance scheduled Gaussians as well as a tensor-specific variance schedule.

A.2 Noise Rejection Sampling

In addition to this coupling, we would need to modify the sampling procedure of the Noising forward process to maintain feasible geometric properties of mesh A, such that the trained reverse process will de-noise the unstructured matrix into a functional structured matrix. This will ensure the "Learned" structuring process both maps to reasonable meshes and does so while maintaining the viability of the mesh.

To accomplish this, we can add a rejection step in each noising process if the addition of the Gaussian noise fails through any of the following points at any time t.

- Results in overly aggressive grading from very small to very large elements size (magnitude cap)
- Produces a "skinny" element where the circumradius is longer than its shortest edge
- Fails to meet the Courant–Friedrichs–Lewy (CFL) condition
- Deteriorates clustering of the matrix eigenspectra
- Produces a large angle element

B Approximating the Hypergradient

Early meta-learning work faced the problem of the unrolled gradient with different approaches: some simply viewed the unrolled Jacobian as the identity matrix so it seems like it just took one really good step (though this misses important model trajectory information), while others would just truncate the gradient at some fixed number of steps.

We elect to go with the approach proposed in [7]. First, we can write the gradient in terms of all appropriate terms:

$$\frac{\partial \mathcal{L}_V}{\partial \mathbf{x}} + \frac{\partial \mathcal{L}_V}{\partial \theta^*}\frac{\partial \theta^*}{\partial \mathbf{x}} \tag{11}$$

The first 2 terms are easily calculable with automatic differentiation software as they do not require unrolled gradients. However, the last term is more tricky. If we allow ourselves to assume that $\frac{\partial \mathcal{L}_T}{\partial \theta^*} = 0$, that is that the inner optimization finds a (not necessarily global) minima, we can leverage Cauchy's Implicit Function Theorem to rewrite the last term in terms of an inverse Hessian:

$$\frac{\partial \theta^*}{\partial \mathbf{x}} = -\left[\frac{\partial^2 \mathcal{L}_T}{\partial \theta \partial \theta^\top}\right]^{-1} \frac{\partial^2 \mathcal{L}_T}{\partial \theta \partial \mathbf{x}^\top} \tag{12}$$

Lastly, the inverse Hessian is problematic as it could be expensive to compute exactly. Instead, we can leverage a Neumann approximation to approximate it:

$$\left[\frac{\partial^2 \mathcal{L}_T}{\partial \theta \partial \theta^\top}\right]^{-1} = \lim_{i \to \infty} \sum_{j=0}^{i} \left[I - \frac{\partial^2 \mathcal{L}_T}{\partial \theta \partial \theta^\top}\right]^j \tag{13}$$

The number of terms we allow the series to have is analogous to how many steps we unroll the inner optimization. The authors found that this method scales well to high numbers of hyperparameters, which is beneficial in our case where the matrix of sensor points are hyperparameters. In addition, the equations above never need to actually compute any Jacobian or Hessian terms fully as they are all used in Jacobian/Hessian-vector products, which modern automatic differentiation software like torch.autograd have fast implementations for.

In our experiments, we found that 3 Neumann series terms was sufficient to get good results, though tuning of this hyperparameter (of the hyperparameter optimizer if it wasn't confusing enough) is left to future work. It could also be interesting to experiment with other ways to solve for an approximate Hessian-inverse product. The original authors compare against the conjugate gradient method, an iterative solver for symmetric, positive definite systems. However, we have no guarantee that the Hessian is positive definite as we cannot guarantee strong convexity of the overall loss; this would motivate the usage of other Krylov solvers like MINRES or the conjugate residual method [11] which do not require positive definite-ness.