XXXI Congresso de Iniciação Científica Unicamp 2023

INTEGRAÇÃO NUMÉRICA DE ALTA PERFORMANCE COM DEEPONETS

Palavras-chave: Computação de alta performance, integração numérica, deep learning, análise em larga escala, Projeto Sirius.

Aline Yasmin Chimbida Guillermo, FEM - Unicamp

Prof. Dr. Josué Labaki, (orientador), FEM, Unicamp

Prof. Dr. Alberto C. Nogueira, (co-orientador), FEM, Unicamp

OBJETIVOS DA PESQUISA

O objetivo deste projeto é utilizar a arquitetura de rede neural DeepONet para integração numérica de alta performance.

O estudo foi motivado pelo Projeto Sirius (fonte de luz síncrotron brasileira). Por ser um acelerador de partículas de larga escala, ele demanda requisitos rigorosos de vibração para operar adequadamente (Liu et al., 2014).

Além disso, os métodos de análise atuais encaram o desafio do alto custo computacional. Isso é porque a interação solo-estrutura usa modelos de discretização de elementos de contorno, os quais demandam integrais computacionalmente caras de funções de Green. Estudos prévios do laboratório desenvolveram modelos cujo tempo de cálculo passava de 3000 anos. A dificuldade de avaliar integrais de funções de Green se deve às singularidades e cauda com decaimento oscilatório, como apresentado na Figura 1.

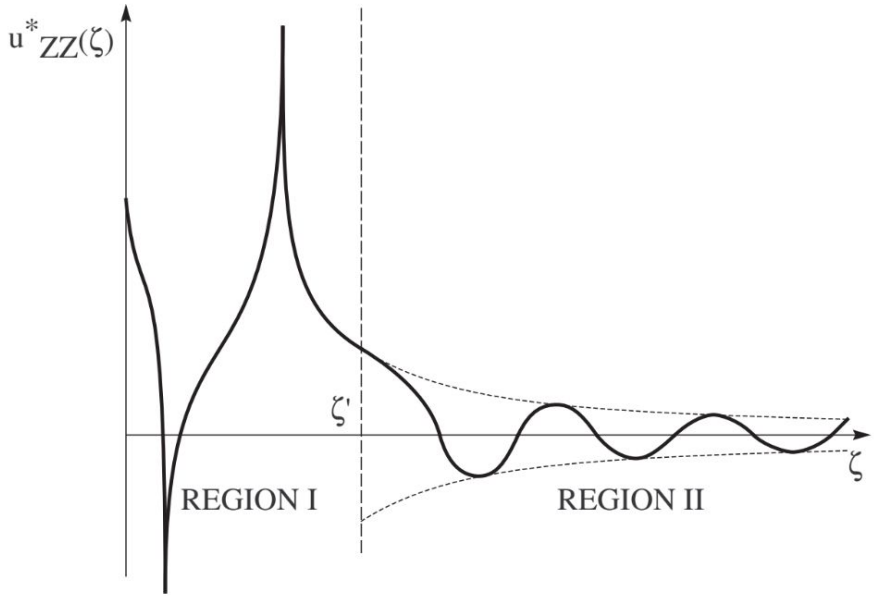

Fig 1: Típico integrando das funções de influência do solo

Assim, a arquitetura de rede neural DeepONet foi escolhida para superar o problema do custo computacional para funções similares às funções de Green. Essa arquitetura foi escolhida devido ao seu potencial de aplicação diversa e boa eficiencia e performance em estudos prévios.

METODOLOGIA DA PESQUISA

A arquitetura DeepONet consiste em duas redes neurais, representadas na Figura 2. A Branch net é responsável por codificar o espaço da função discreta de input. A segunda rede é a Trunk net, que codifica o domínio das funções de output.

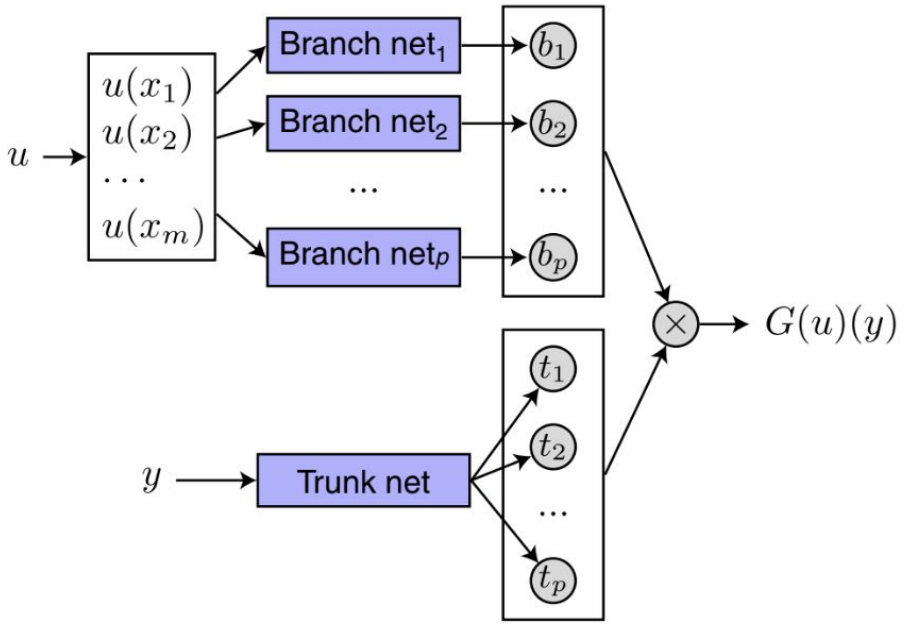

Fig 2: Representação da arquitetura da DeepONet. Fonte: Lulu et al., 2021.

O problema consiste em aprender o operador de antiderivada G : v → u definido pela equação diferencial ordinária com condição inicial u(0) = 0:

$$\frac{du(x)}{dx} = v(x), \quad x \in [0, 1]$$

Uma das abordagens foi utilizar a branch net com dimensão (1, m) contendo os valores da função de input. “m” é o número de pontos utilizados na discretização da função. E a trunk net é um array de dimensão (m, 1) contendo os pares dos valores de input no eixo x. E o output da rede é um vetor de dimensão (m, 1) que possui os valores de antiderivadas calculada para cada ponto.

Para este caso, a Figura 3 exemplifica como são os conjuntos de teste e treino gerados:

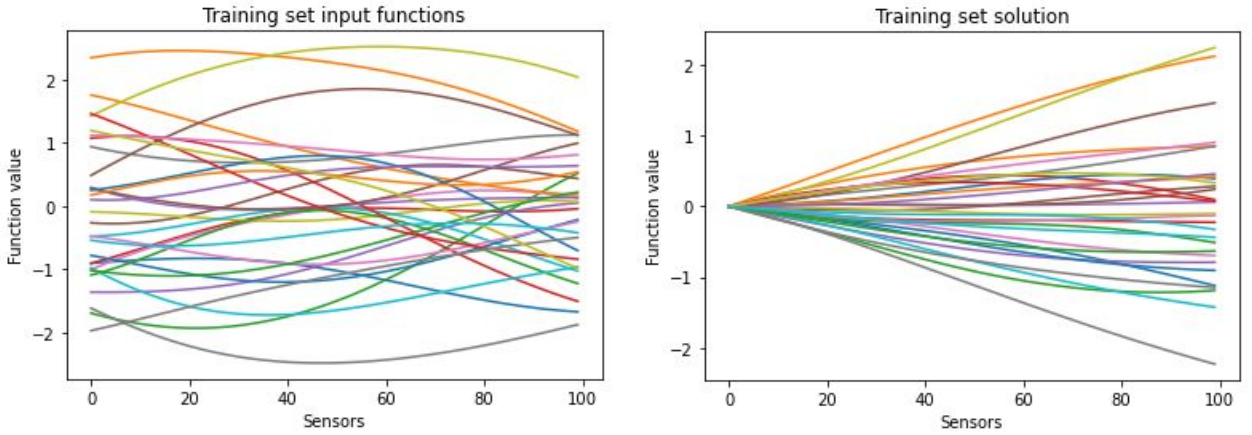

Fig 3: Curvas para treino e teste da rede.

Outra forma de setar a rede para o caso de funções polinomiais foi definir os dados com base nos coeficientes polinomiais ao invés de vetores de pontos. Assim, a branch net é composta pelos coeficientes, a trunk net é formada pela coordenada x onde a antiderivada é calculada. E o output da rede é o valor da antiderivada.

RESULTADOS E DISCUSSÃO

A figura 4 mostra o resultado da previsão das integrais das funções polinomial e exponencial calculadas pela DeepONet. A curva de “input” representa a função do integrando que é submetida à branch net. A curva “expected” representa a solução analítica da integração da função de input. E a curva “antiderivative” é a solução calculada pela DeepONet.

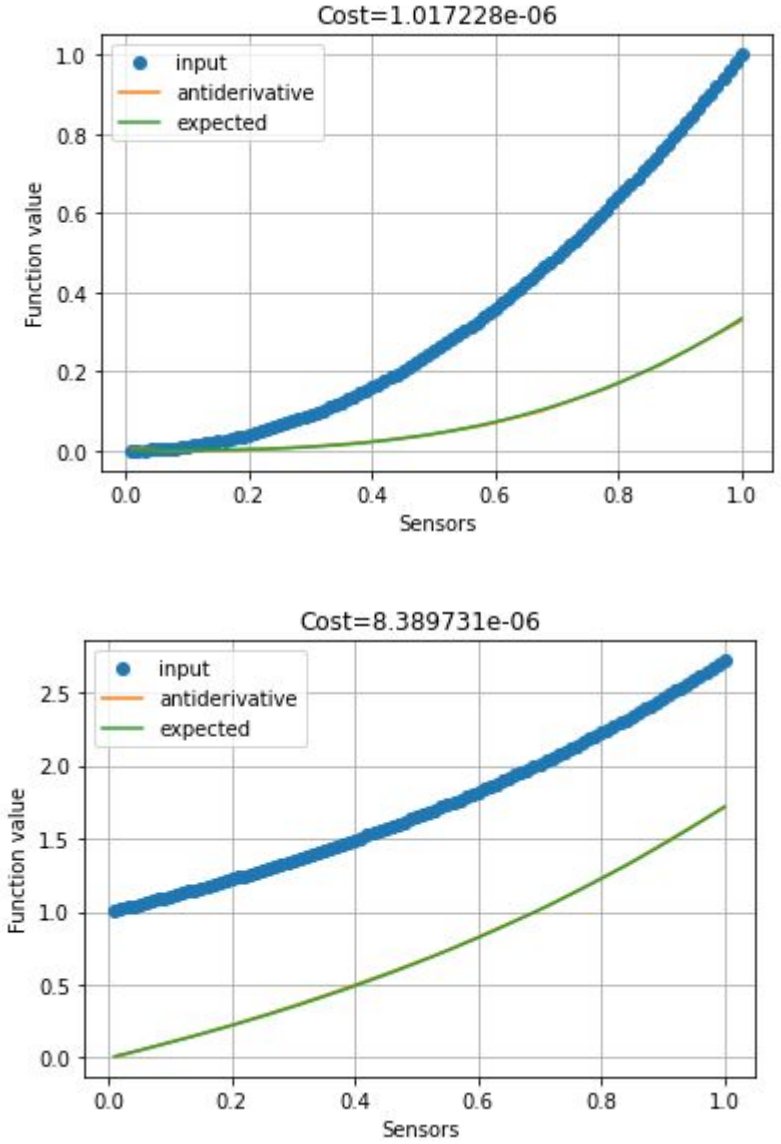

Fig 4: Comparação da antiderivada esperada e calculada para as funções u(x) = x^2 e u(x) = e^x.

O valor "cost" apresentado nos gráficos é um valor proporcional à média do quadrado da diferença entre a solução analítica e calculada ponto a ponto.

$$\text{Cost} = \frac{1}{2m}\sum_{i=1}^{m}(y_i - \hat{y})^2$$

Para estes casos, nota-se que a rede neural foi capaz de predizer a antiderivada com boa acurácia, com custo baixo na ordem de 10E-06.

O próximo passo foi avaliar a precisão do cálculo de integrais de funções oscilatórias. Foram testadas funções seno com diferentes períodos e discretizações (pontos de amostra da função). Observou-se que a rede teve mais dificuldade para calcular funções com período menor e pouca discretização, onde o custo foi da ordem de 10E-02.

Para mitigar este problema, foi necessário adicionar funções mais representativas com características oscilatórias nos dados de treino da rede. Observe na Figura 5 a comparação entre os resultados antes e depois do refinamento das funções de treino. Além da melhora no custo, com diminuição de 2 ordens de grandeza, nota-se que a curva calculada aproxima-se visualmente melhor da solução analítica.

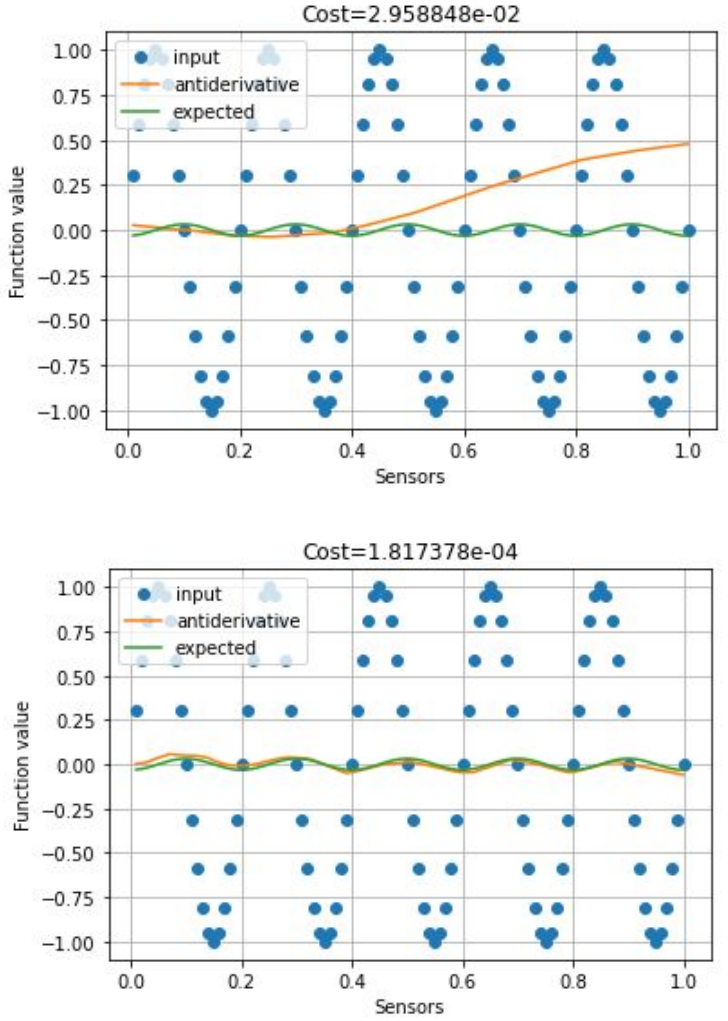

Fig 5: Comparação da antiderivada esperada e calculada para as funções oscilatórias antes (acima) e depois (abaixo) do refinamento do conjunto de treino.

Outro tipo de função avaliado foram as funções com singularidades. Neste caso a DeepONet apresentou dificuldade de avaliar as integrais com precisão nas regiões de singularidade com grande diferença de ordem de grandeza na função de input (Figura 6).

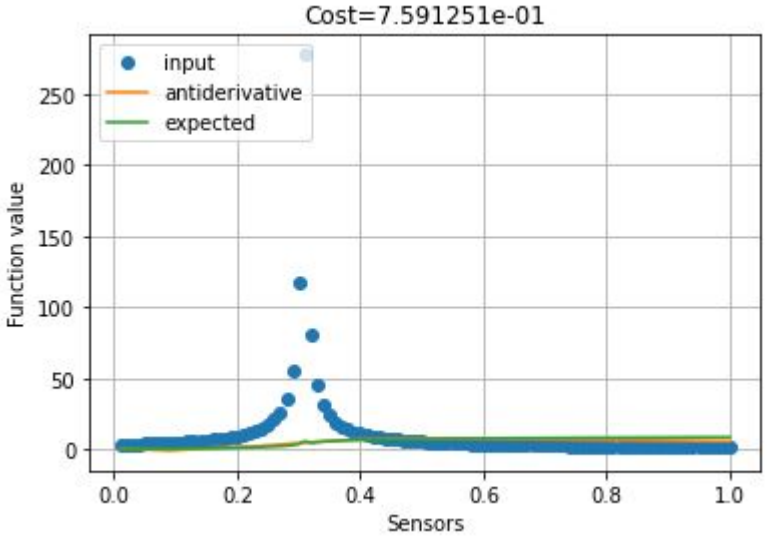

Fig 6: Resultado da DeepONet para função com singularidade.

Um fator que melhorou a acurácia dos resultados foi acrescentar um fator de amortecimento da singularidade, assim a nova forma da função ficou:

$$f(x) = \frac{1}{x(1+\eta i) - a}$$

Onde η é o fator de amortecimento, "i" é o número complexo e "a" é a posição da singularidade

O resultado da função amortecida está apresentado na Figura 7.

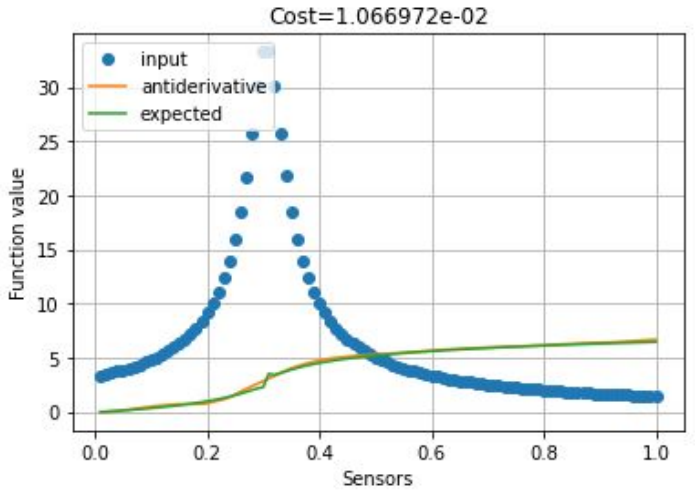

Fig 7: Resultado da DeepONet para função singular com fator de amortecimento η = 0,1.

A dificuldade da DeepONet de lidar com diferentes ordens de magnitude na função e input já era um problema intrínseco conhecido dessa arquitetura (Wang et al., 2021). Uma sugestão de melhora foi treinar a rede utilizando uma função perda normalizada. Porém foi observado que o novo método não melhorou o resultado.

O próximo tópico abordado foi retornar ao estudo das funções polinomiais com o objetivo de entender se essa arquitetura tem melhor desempenho com outros tipos de integrandos.

Desta forma, utilizou-se os indicadores de "cost" e "error" para avaliar o impacto que a modificação dos parâmetros tem nos resultados.

$$\text{Error} = \frac{\left|y_i - \hat{y}\right|}{\left|\hat{y}\right|} * 100\%$$

Além disso, para cada experimento foi coletado o tempo de treinamento da rede para comparar o custo computacional de cada variação de parâmetro.

Os parâmetros variados foram:

- Tamanho da matriz com os dados de treinamento
- Número de camadas da rede
- Função de ativação
- Taxa de aprendizado inicial
- Número de iterações
- Número de sensores

Cada uma das modificações sozinha melhorou o erro relativo no máximo até a ordem de grandeza de 10E-03 e o custo 10E-06.

Outra observação feita foi a de que o resultado do erro independe do ponto da curva onde a diferença é calculada. Ou seja, no caso de funções não singulares, não faz diferença calcular a integral de interesse no meio ou final da curva.

Por fim, mesmo aplicando as melhores combinações de parâmetros testados, o resultado do erro foi de 10E-03. O adequado seria alcançar um erro em torno de 10E-06 para ser útil no campo da engenharia.

Uma vez que o erro não ficou suficientemente pequeno, implementou-se outra abordagem para se definir os

dados de input da rede. Ao invés de utilizar um vetor de pontos que representam uma curva de função, o input agora é definido diretamente pelos coeficientes do polinômio e a posição de interesse para calcular a integral no eixo x .

Os vetores de treino e teste possuem os mesmos coeficientes que o polinômio de interesse com variadas posições de eixo x e as antiderivadas correspondentes. A solução se tornou um caso de interpolação de dados.

Isso simplificou o problema, diminuiu a quantidade de dados e com isso melhorou a eficiência computacional do treino da rede.

O novo método apresentou melhor acurácia e melhor eficiência comparado ao método anterior, utilizando parâmetros similares.

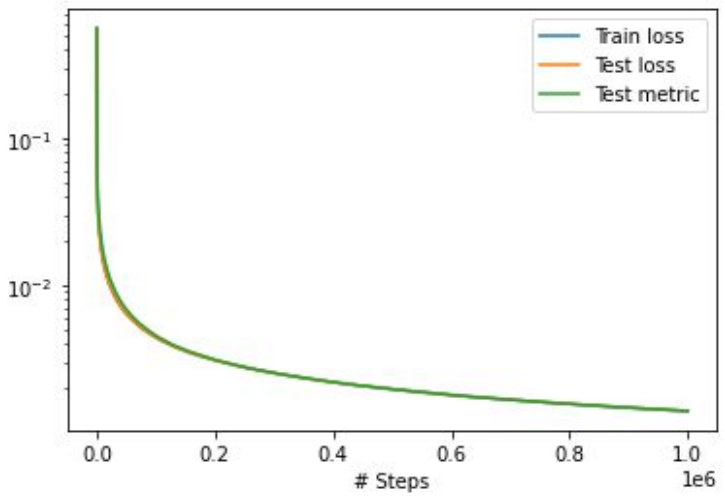

Fig 8: Método da curva discretizada. Tempo de treino: 1383,3s. Erro relativo: 0,17%.1.

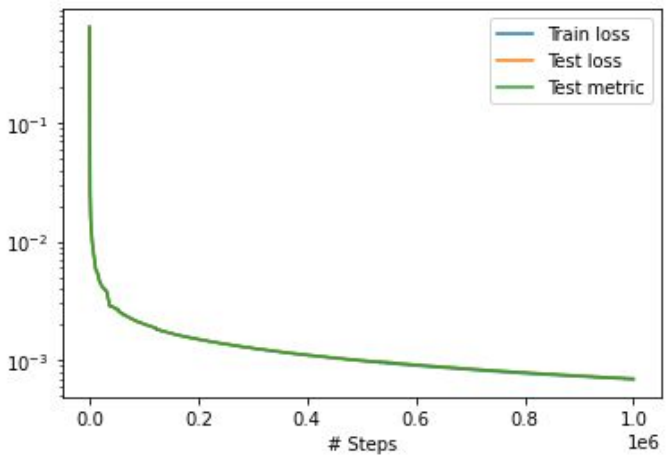

Fig 9: Novo método dos coeficientes. Tempo de treino: 460,1s. Erro relativo: 0,082%.

A tabela abaixo contém uma comparação de eficiência computacional e acurácia entre a melhor versão da DeepONet refinada neste estudo e o MATLAB. Observe que a DeepONet se destaca na eficiência computacional, com o tempo de cálculo de uma integral mais baixo.

	MATLAB polyint (solução analitica)	MATLAB integral (solução numérica)	DeepONet (10e7 iterations)
Tempo médio para calcular uma integral (s)	0,00629	0,03364	0,00040
erro relativo (%)	-	0,00%	0,01%
Tempo de treino (h)	-	-	2,1

BIBLIOGRAFIA

Labaki, J., Barros, P. L., and Mesquita, E. (2021). A model of the time-harmonic torsional response of piled plates using an ibem-fem coupling. Engineering Analysis with Boundary Elements, 125:241–249.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. nature, 521(7553):436.

Liu, L., Milas, N., Mukai, A. H., Resende, X. R., and de Sá, F. H. (2014). The sirius project. Journal of synchrotron radiation, 21(5):904–911.

Lu, L., Jin, P., Pang, G., Zhang, Z., and Karniadakis, G. E. (2021). Learning nonlinear operators via deeponet based on the universal approximation theorem of operators. Nature Machine Intelligence, 3(3):218–229.

Rawat, W. and Wang, Z. (2017). Deep convolutional neural networks for image classification: A comprehensive review. Neural computation,29(9):2352–2449.

Wynn, P. (1961). The epsilon algorithm and operational formulas of numerical analysis. Mathematics of Computation, 15(74):151–158.

Nielsen, Michael (2018). Neural networks and Deep Learning. Determination Press. Retrieved from http://neuralnetworksanddeeplearning.com/

Stewart, James. Calculus, Volume I. São Paulo: Cengage Learning, 7th edition, 2013

Wang, S., Wang, H., and Perdikaris, P. (2021). Improved architectures and training algorithms for deep operator networks. Cornell University. arXiv:2110.01654