Lecture 8: Multi-stage Neural Networks

Instructor: Wenxiao Pan

1 Introduction

Neural networks have been proven to be universal function approximators [1, 2]. However, in practice, neural network training often falls into local minima [3, 4], causing the training loss to plateau even after many epochs' training. Numerous methods in literature have focused on different aspects, such as activation function selection [5, 6], network configuration [7–9], optimization techniques [10, 11], trainable weights [12], and loss function [13, 14], effectively enhancing the convergence rate of the loss function for various problems. However, few of these methods manage to reduce the training error less than $O(10^{-5})$, even with large network size and extended training iterations. In contrast, **classical numerical methods (e.g., finite difference)** can systematically enhance solution's accuracy by simply reducing the grid size. This is a major shortcoming of neural networks for solving many problems within mathematical and physical sciences.

The idea of multi-stage neural networks is to divide the training process into different stages, with each stage using a new network that is optimized to fit the residue from the previous stage. The multi-stage neural networks can effectively mitigate the spectral biases associated with regular neural networks, enabling them to capture the high frequency feature of target functions. As a result, the combined neural networks obtained from different stages can approximate the target function with remarkably high accuracy.

2 Methodology

For illustration, we consider regression problems that involve predicting a continuous output variable u as a function of the input variable x. We train a neural network that represents $u(x)$ to fit N_d data points, denoted (x_i, u_i). The loss function for a regression problem is typically the mean squared error (MSE), defined as

$$\mathcal{L} = \frac{1}{2N_d}\sum_{i=1}^{N_d}[u(x_i) - u_i]^2 \,. \tag{1}$$

2.1 Limitation of regular neural network training

We consider a target function:

$$u_g(x) = \sin(2x+1) + 0.2e^{1.3x} \,. \tag{2}$$

1. We generate training data by sampling 300 data points from it without noise, uniformly distributed within the domain $x \in [-1, 1]$.
2. To fit the training data, we use a fully-connected neural network made of 3 hidden layers each with 20 units and use **hyperbolic tangent** (tanh) as the activation function.

Figure 1(a) shows that the trained neural network $u_0(x)$ captures the target function $u_g(x)$ well. During the iterations, the training loss $\mathcal{L}$ based on MSE between the data and network, decreases significantly at the early stage (Figure 1(d)). However, after 5000 iterations, it reaches a plateau around $O(10^{-7})$ with very small convergence rate. **The plateau of training loss corresponds to a mismatch between the trained network $u_0(x)$ and target function $u_g(x)$ at high frequencies.** Figure 1(b) demonstrates that the error function $e_1(x) = u_g(x) - u_0(x)$ within the domain is indeed a high-frequency function. As known, a standard multi-layer neural network struggles to learn the high frequencies of a target function.

Further experiments affirm that this plateau value of the error remains consistent even with larger networks and additional data. Figure 2 show that the root mean square value (RMS) ϵ of the error $e(x)$ between the trained network $u_0(x)$ and the data from the target function $u_g(x)$ remains unchanged even when the number of either neurons (Figure 2(a)) or layers (Figure 2(b)) is increased. Although the RMS error ϵ does slightly decrease with an increase in training data (Figure 2(c)), **this reduction is smaller than the standard deviation of eight repetitive**

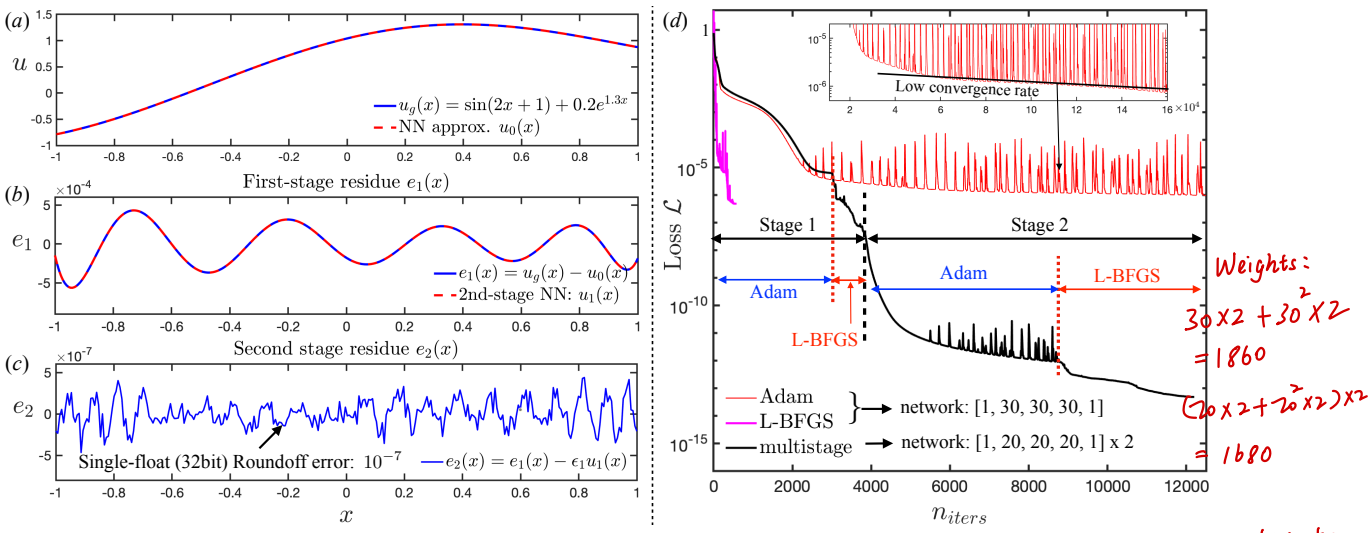

Figure 1: Single-stage training vs. multi-stage training.

experiments conducted with different random initializations, and thus is negligible. These results suggest that the plateau in training loss is not due to insufficient neural network size or lack of training data, but instead arises from inherent limitations of the training process itself.

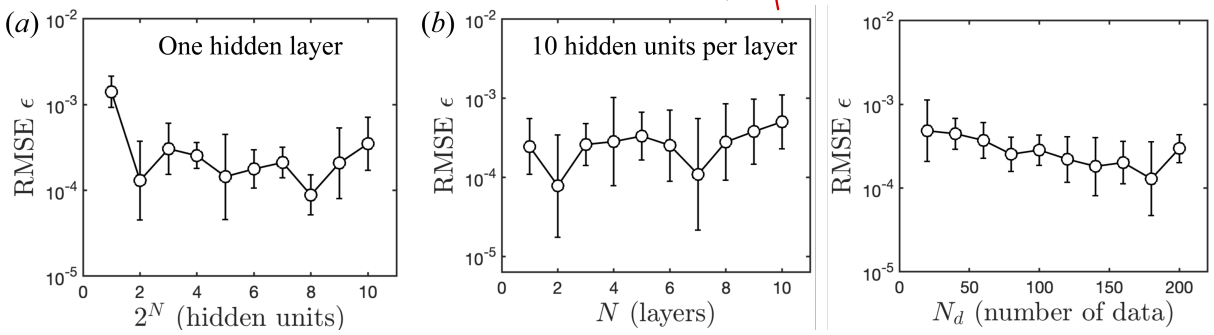

Figure 2: Root mean square (RMS) value ϵ of the error between the target function and trained network using different number of (a) hidden units (neurons), (b) layers, and (c) training data. Error bars show the standard deviation of eight repetitive experiments with different random initialization.

2.2 Key settings of multi-stage training scheme

The original training data from (2) is denoted with $(x^{(i)}, u_g^{(i)})$. The training data for the second neural network would be $(x^{(i)}, e_1(x^{(i)}))$, where $e_1(x^{(i)}) = u_g(x^{(i)}) - u_0(x^{(i)})$ denotes the error of network at $x^{(i)}$. At this point, extra care should be taken when setting up the second neural network, particularly concerning two key aspects.

2.2.1 Magnitude of the second neural network

While the original training data has a magnitude of $O(1)$, the training data for the second neural network, $e_1(x^{(i)})$, would be much smaller than 1 (as shown in Figure 1(b)). A neural network employing regular weight initialization methods, such as Xavier [**?**], often struggles to capture training data whose magnitude is significantly larger or smaller than 1.

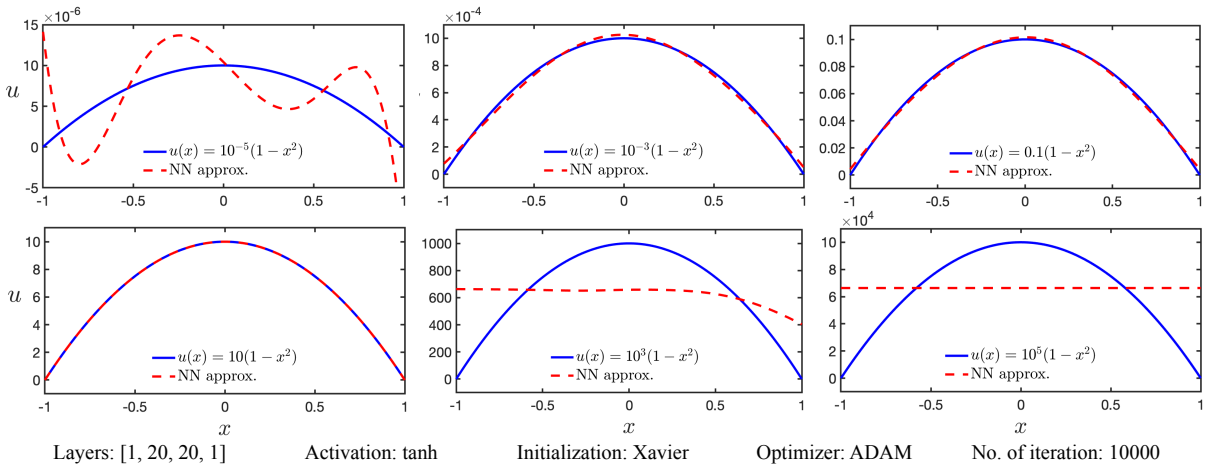

Figure 3: Fitting of neural networks to the data with different magnitudes without normalization. It shows that the network is hard to fit data with magnitude either too much larger or smaller than 1.

A straightforward solution to this issue is to normalize the training data by its root mean square value ϵ_1, defined as

$$\epsilon_1 = \sqrt{\frac{1}{N_d}\sum_{i=1}^{N_d}[e_1(x^{(i)})]^2} = \sqrt{\frac{1}{N_d}\sum_{i=1}^{N_d}[u_g^{(i)} - u_0(x^{(i)})]^2}\,. \tag{3}$$

Thus, the normalized training data for the second neural network becomes $(x^{(i)}, e_1(x^{(i)})/\epsilon_1)$. **Denoting the second trained network as $u_1(x)$**, the combined networks become

$$u_c^{(1)}(x) = u_0(x) + \epsilon_1 u_1(x)\,. \tag{4}$$

Subsequently, one can continue training the third or even further neural networks to reach higher accuracy. The training data for the $(n+1)$-th neural network u_n is the error e_n normalized by its own magnitude (root mean square value) ϵ_n, namely $(x^{(i)}, e_n(x^{(i)})/\epsilon_n)$, where

$$\epsilon_n = \sqrt{\frac{1}{N_d}\sum_{i=1}^{N_d}[e_n(x^{(i)})]^2}\,, \tag{5}$$

$$e_n(x) = e_{n-1}(x) - \epsilon_{n-1}u_{n-1}(x) = u_g(x) - \sum_{j=0}^{n}\epsilon_j u_j(x)\,, \tag{6}$$

with $\epsilon_0 = 1$. Finally, the ultimate trained model that combines all the $(n+1)$ neural networks reads:

$$u_c^{(n)}(x) = \sum_{j=0}^{n}\epsilon_j u_j(x)\,. \tag{7}$$

2.2.2 Fitting high-frequency data

Even with normalization, the second neural network, if initialized with regular weights, could still struggle to fit the high-frequency data due to the inherent spectral biases of neural networks. Following the discussion in the last lecture: "**Deep Neural Networks-Spectral Biases**", we build the second neural network with the scale factor $\hat{\kappa}$ obtained from the dominant frequency f_d of the error $e_1(x)$.

Figure 1(b) shows the result of the second-stage training, which fits the residue $e(x)$ between the first-trained network $u_0(x)$ and the original training data $u_g(x)$. In comparison to the single-stage training where the loss plateaued $O(10^{-8})$, the two-stage training dramatically reduced loss to $O(10^{-14})$ (Figure 1(d)), resulting in the fitting error $e(x)$ between the two-stage trained networks and the data reaches $O(10^{-7})$ (Figure 1(c)).

Moreover, we note that **the total number of weights** used in the two-stage neural networks, $2 \times (2 \times 20^2 + 2 \times 20)) = 1680$, is less than that in the single-stage network, $2 \times 30^2 + 2 \times 30 = 1860$. This underscores that a

larger neural network is not inherently advantageous; an appropriate training scheme is more vital and efficient for the reduction of training loss.

Remark: Figure 1(d) also presents the performance of two different optimizers: Adam [**?**], a first-order stochastic gradient descent method, and L-BFGS [**?**], a second-order quasi-Newton method. Training with L-BFGS quickly converges, but the loss reaches the same plateau as Adam. This suggests that the loss plateau is not optimizer-specific. Combining them, e.g., first Adam and then L-BFGS, can help in some degree reducing the training loss, but the main reduction is due to the multi-stage training scheme.

References

[1] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[2] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural networks*, 3(5):551–560, 1990.

[3] Pierre Baldi and Kurt Hornik. Neural networks and principal component analysis: Learning from examples without local minima. *Neural networks*, 2(1):53–58, 1989.

[4] Aditi Krishnapriyan, Amir Gholami, Shandian Zhe, Robert Kirby, and Michael W Mahoney. Characterizing possible failure modes in physics-informed neural networks. *Advances in Neural Information Processing Systems*, 34:26548–26560, 2021.

[5] Vincent Sitzmann, Julien Martel, Alexander Bergman, David Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. *Advances in Neural Information Processing Systems*, 33:7462–7473, 2020.

[6] Vishwanath Saragadam, Daniel LeJeune, Jasper Tan, Guha Balakrishnan, Ashok Veeraraghavan, and Richard G Baraniuk. Wire: Wavelet implicit neural representations. *arXiv preprint arXiv:2301.05187*, 2023.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[8] Ameya D Jagtap and George E Karniadakis. Extended physics-informed neural networks (xpinns): A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations. In *AAAI Spring Symposium: MLPS*, pages 2002–2041, 2021.

[9] Ben Moseley, Andrew Markham, and Tarje Nissen-Meyer. Finite basis physics-informed neural networks (fbpinns): a scalable domain decomposition approach for solving differential equations. *arXiv preprint arXiv:2107.07871*, 2021.

[10] Chun-Chen Tu, Paishun Ting, Pin-Yu Chen, Sijia Liu, Huan Zhang, Jinfeng Yi, Cho-Jui Hsieh, and Shin-Ming Cheng. Autozoom: Autoencoder-based zeroth order optimization method for attacking black-box neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 742–749, 2019.

[11] Pao-Hsiung Chiu, Jian Cheng Wong, Chinchun Ooi, My Ha Dao, and Yew-Soon Ong. Can-pinn: A fast physics-informed neural network based on coupled-automatic–numerical differentiation method. *Computer Methods in Applied Mechanics and Engineering*, 395:114909, 2022.

[12] Levi McClenny and Ulisses Braga-Neto. Self-adaptive physics-informed neural networks using a soft attention mechanism. *arXiv preprint arXiv:2009.04544*, 2020.

[13] Mingchen Li, Xuechen Zhang, Christos Thrampoulidis, Jiasi Chen, and Samet Oymak. Autobalance: Optimized loss functions for imbalanced data. *Advances in Neural Information Processing Systems*, 34:3163–3177, 2021.

[14] Remco van der Meer, Cornelis W Oosterlee, and Anastasia Borovykh. Optimally weighted loss functions for solving pdes with neural networks. *Journal of Computational and Applied Mathematics*, 405:113887, 2022.