Machine Learning
Assignment 3

CHANG Yung-Hsuan (張永璿)
111652004
eiken.sc11@nycu.edu.tw

September 23, 2025

1. Read “On the Approximation of Functions by tanh Neural Networks” and focus on Lemma 3.1 and Lemma 3.2. Write a report that explains the statements and ideas behind these two lemmas. Your explanation should be written so that a college student who has completed Calculus I and II can understand. Avoid unnecessary technical jargon—your goal is to make the arguments accessible and clear.

   Solution. First, as the question requires “your explanation should be written so that a college student who has completed Calculus I and II can understand,” I explain the background of neural networks.

   A neural network can be thought of as a flexible mathematical machine that takes an input (like a number x) and produces an output (it may be a number, a vector, or even a matrix) through a combination of weighted sums and nonlinear functions. In this paper, the nonlinear function (it’s officially called the activation function) used is the hyperbolic tangent tanh, which maps real numbers to $(-1, 1)$. You can think of it *kind of* like the inverse tangent but with more beautiful properties and nice range.

   A neural network consists of three parts: an input layer, hidden layers, and an output layer. The input layer merely takes the input, and the output layer produces the output. Their dimension (number of neurons) depends on the input and desired output. The interesting part is hidden layers.

There can be more than one layers in the hidden layers, and the number of neurons can be either less than, equal to, or more than the input layer or the output layer.

The central idea in approximation theory is that neural networks can mimic almost every function. This paper focuses on one-hidden-layer neural networks, and these two lemmas focus on showing how (with how many neurons in the single hidden layer) one-hidden-layer neural networks with tanh as activation functions can approximate monomials, which are the building blocks of polynomials.

Now, I explain the statements and ideas behind these two lemmas.

Lemma 3.1. This lemma states that all odd-power monomials (that is, $x, x^3, x^5, ..., x^s$ when s is odd) can be approximated by a one-hidden-layer neural network with $\frac{s+1}{2}$ neurons. The symbol s means the maximum degree of the monomial we want to approximate, and the result also guarantees accuracy not only for the function itself but also for its derivatives up to order k. In other words, the approximation is uniformly good both for the function values and their slopes, curvatures, and so on.

The constructive idea is to begin with low-degree cases and move upward: to approximate x^{2n+1}, you only need one more neuron beyond what is required for x^{2n-1}, and all the neurons used for x^{2n-1} can also be reused in the higher-degree case. This means that the number of neurons grows steadily as the degree grows.

The reason this works well for odd powers is that tanh is itself an odd function. By combining shifted and scaled versions of tanh, one can cancel lower-order terms and isolate the desired odd monomial. As a result, a one-hidden-layer network can be tuned to produce curves that closely resemble x^3 or x^5 on the interval of interest.

Lemma 3.2. This lemma covers even-power monomials (that is, $x^2, x^4, x^6, ..., x^s$ when s is even). Since tanh is odd, it cannot directly approximate even functions. The key trick is to express even powers in terms of odd powers using algebraic identities. For example,

$$x^2 = \frac{(x+1)^3 - (x-1)^3}{6}.$$

This shows that even powers can be written as linear combinations of odd powers. Once odd powers are available (by Lemma 3.1), even powers can be constructed by combining them. The cost is that the network requires more neurons, $\frac{3(s+1)}{2}$ instead of $\frac{s+1}{2}$, but it still remains a single hidden layer.

Conclusion. With Lemma 3.1 handling odd powers and Lemma 3.2 handling even powers, we can approximate all monomials up to degree s within a neural network. Since polynomials are sums of monomials, this means a one-hidden-layer neural network with activation functions tanh can approximate any polynomial, while also controlling derivatives up to order k.

2. There are unanswered questions from the lecture, and there are likely more questions we haven't covered. Take a moment to think about these questions. Write down the ones you find important, confusing, or interesting.

   Answer. Will we investigate CNN in this class? For example, how does convolution work effectively?

3. Use exactly the same code you used in Assignment 2 - Programming Assignment 1 and calculate the error in approximating the derivative of the function.

   Solution. I used the same code with defining the true derivative of the Runge function and the derivative of the neural network.

   As the code is the same, a neural network is used with one input layer with one feature, two hidden layers with 64 neurons each, and one output layer with a neuron. The dataset consists of 100 evenly distributed data points with `train_test_split` (with `random_state=4`) to training set 70%, validation set 15%, and testing set 15%. I use tanh as the activation function, and MSE is the loss function, and the Adam is the optimizer.

   The following table presents the errors.

	$\sqrt{\text{MSE}}$	Absolute Max Error
Training	$7.19 \cdot 10^{-2}$	$2.54 \cdot 10^{-1}$
Testing	$6.72 \cdot 10^{-2}$	$1.94 \cdot 10^{-1}$

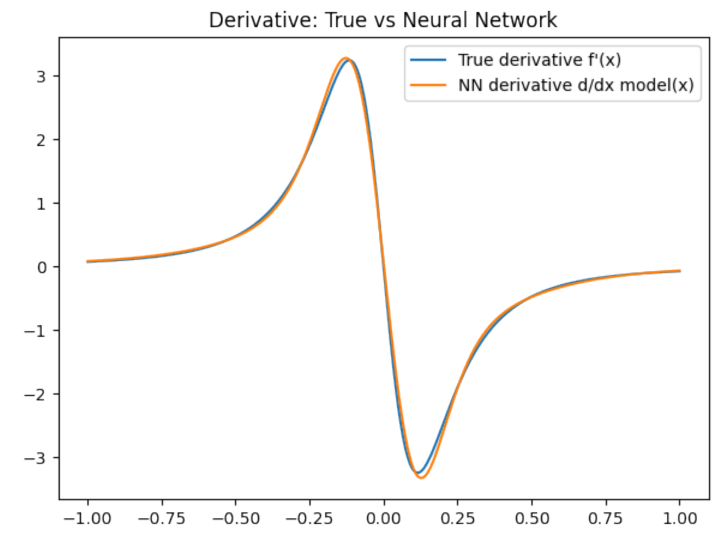

Figure 1. The Derivative of the Runge Function and the Derivative of the Neural Network

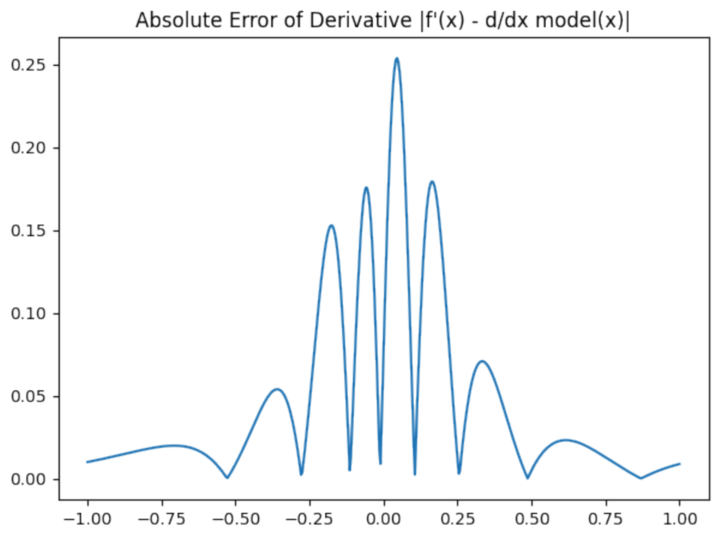

Figure 2. The Absolute Error between the True Derivative and the Derivative of the Neural Network

Please see the GitHub repository for my raw code.

4. Use a neural network to approximate both the Runge function and its derivative. Your task is to train a neural network that approximates

   a. the function $f(x)$ itself, and

b. the derivative $f'(x)$.

Define a loss function consisting of two components,

a. function loss, and

b. derivative loss.

Write a short report (1–2 pages) explaining method, results, and discussion including

a. plot the true function and the neural network prediction together,

b. show the training/validation loss curves, and

c. compute and report errors (MSE or max error).

Solution. I modified the original code above to solve this problem. The main idea is still the same. A neural network is used with one input layer with one feature, two hidden layers with 64 neurons each, and one output layer with a neuron. The dataset consists of 100 evenly distributed data points with `train_test_split` (with `random_state=4`) to training set 70%, validation set 15%, and testing set 15%. I use tanh as the activation function, and MSE is the loss function, and the Adam is the optimizer. However, the loss function here is different; it is defined by

$$L(\theta) = \frac{1}{N} \cdot \sum_{i=1}^{N} \left((f(x_i) - H_\theta(x_i))^2 + \lambda \cdot \left(f'(x_i) - \frac{\mathrm{d}}{\mathrm{d}x} H_\theta(x_i) \right)^2 \right), \tag{1}$$

where λ is balances the contribution of the function loss and the derivative loss. The derivative of the neural network output with respect to x is computed using TensorFlow's automatic differentiation (`tf.GradientTape`). I simply set λ to 1.

The following table presents the errors.

	$\sqrt{\text{MSE}}$	Absolute Max Error
Testing (Function)	$8.98 \cdot 10^{-4}$	$2.03 \cdot 10^{-3}$
Testing (Derivative)	$1.4 \cdot 10^{-2}$	$3.71 \cdot 10^{-2}$

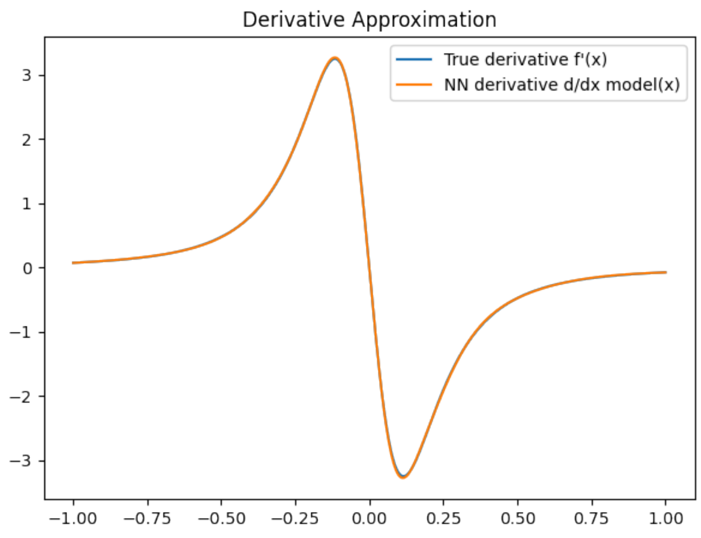

Figure 3. The Derivative of the Runge Function and the Derivative of the Neural Network with Loss Function (1)

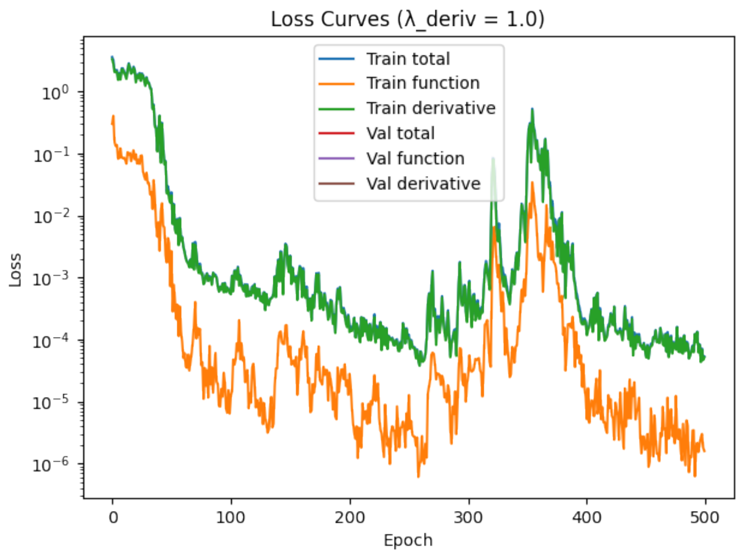

Figure 4. The Loss Curves during the Training Process of the Neural Network with Loss Function (1)

Adding the derivative term to the loss function improves the accuracy of derivative prediction and the smoothness of the function approximation.

For the accuracy of the prediction of the derivative, compare Figure 1 and Figure 3; for the smoothness of the function approximation, compare Figure 5 and Figure 1 in my solution to Assignment 2.

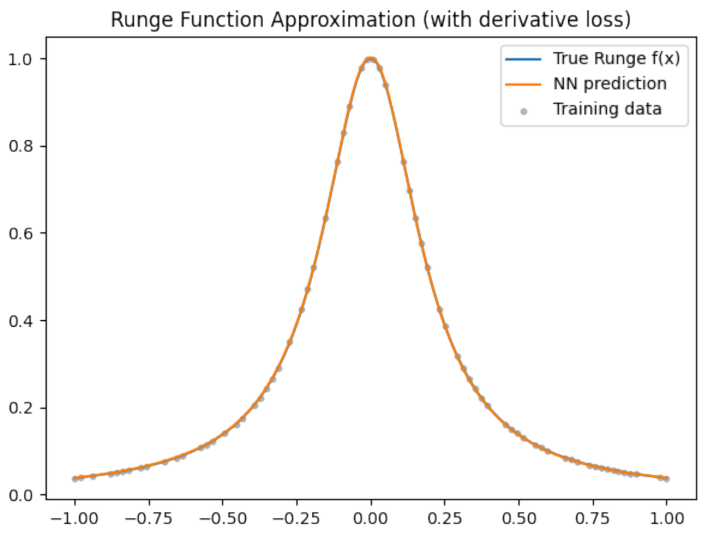

Figure 5. Runge Function Approximation with Neural Network with Loss Function (1)

Please see the GitHub repository for my raw code.