Assignment 3

Written assignment

1.

First of all, I'm going to instroduce some basic idea of neural network.

A neural network can be thought of as a function approximator. The purpose of a neural network is to find the function to match the real value $f(x)$ with the given input x.

Then, the form of the neural network is the composition of affine function and activation function.
Recall, an affine function is of the form $g(x) = Ax + b$, where A is the weights and b is bia. And the activation function $\sigma(x)$ is a nonlinear function, like relu, sigmoid, tanh and so on. The reason why we composite activation function with affine function is to make the network to bend, curve, and approximate complicated shape, since the composition of affine function will only produce a linear function.

We defined the neural network as:

$$h(x) = g_l \circ \sigma \circ g_{l-1} \circ \sigma \circ \cdots \circ \sigma \circ g_1$$

where, g_1 is the input layer, g_l is the output layer and g_k, where $k = 2, \ldots, l-1$ is the hidden layer.

Learning just means adjusting the weights of A and biases b so the output gets closer to the desired value(the real function value).

Then, I'm going to explain the statments and ideas behind the 2 lemmas.

Lemma 3.1

This lemma states that the neural networks are accurate approximation to monomails with odd degree.($x, x^3, \ldots, x^s$, where s is odd). And the neural network contains one-hidden-layer with $\frac{s+1}{2}$ neurons, s is the degree of the

monomial we want to approximate. This approximation is good on both the function values' approximation and their slopes, curvatures.

This idea begins with low-degree case and extend to approximate x^{2n+1}, the iconic feature of this approximation is that you only need one more neuron in contrast to x^{2n-1}, and all the neuron used to approximate x^2n-1 can also be used in the higher degree odd monomials. This makes the penalty lower while the degree grows while we are approximating.

The reaon why this methods works well for odd degree monomials is that $tanh$ is an odd function. By combining shifted and scaled versions of $tanh$, we can cancel the lower-degree terms and isolated the desired odd monomials. While only combining shifted and scaled $tanh$, we fulfill this with one-hidden-layer. And the produced curves can closely resemble x^3, x^5.

Lemma 3.2

This lemma stats even degree monomials.($x^2, x^4, \ldots, x^s$, where s is even). Here we cannot approximate even functions directly as **Lemma3.1** since $tanh$ is an odd function. So, we need to express even degree monomials in terms of odd degree one with algebraic identities. Take x^2 for example:

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

This shows that even degree monomials can be written as linear combination of odd degree monomials. Once the odd degree monomials can be approximated by **Lemma3.1**, even one can be constructed by combining them. By doing these, the neural network requres more neurons, $\frac{3(s+1)}{2}$ neurons, and still requires one hidden layer.

2.

Will the derivation of the monomials performs as well as the orginal functions with this methods?
Why should we choose $tanh$ as the activation function? Can we choose a smooth even function to approximate the monomials? Will it perform better than $tanh$ in approximating monomials?

Program assignment

1

Here, I use the same code `runge.py` to approxinate the derivative of Runge function.

I change the def of runge function to runge_prime.
Hence, I can use this code to approximate it.

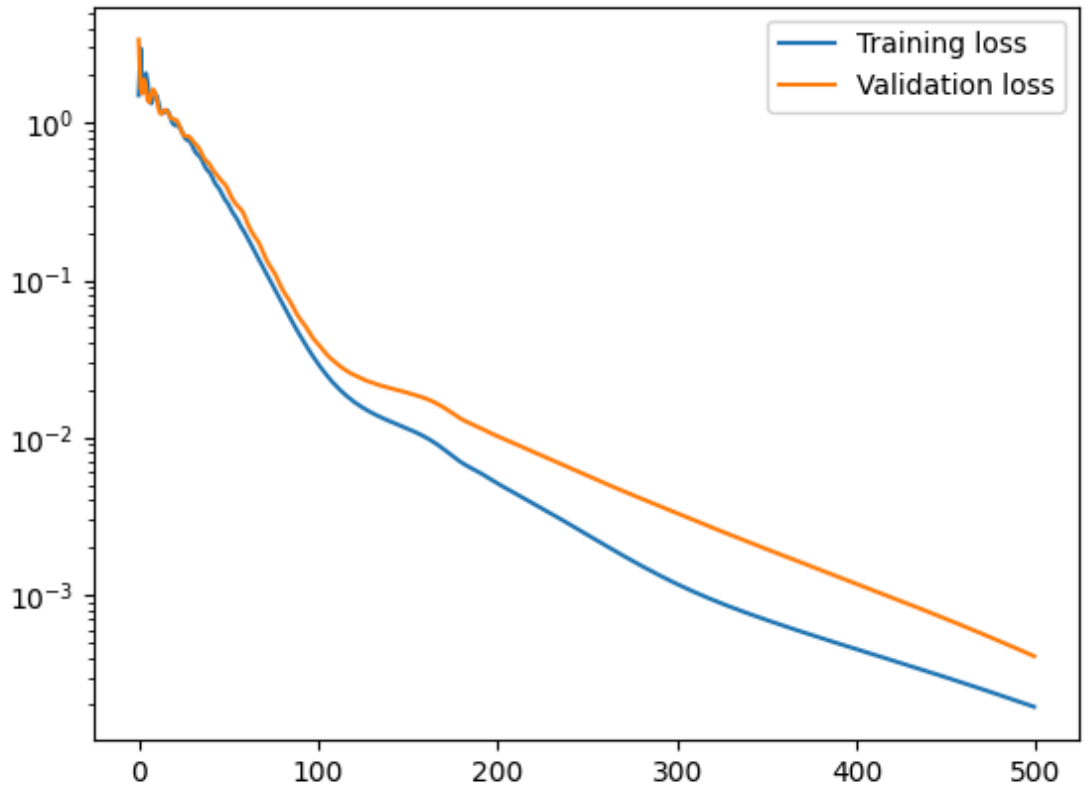

Figure 1. The loss curve of runge prime.

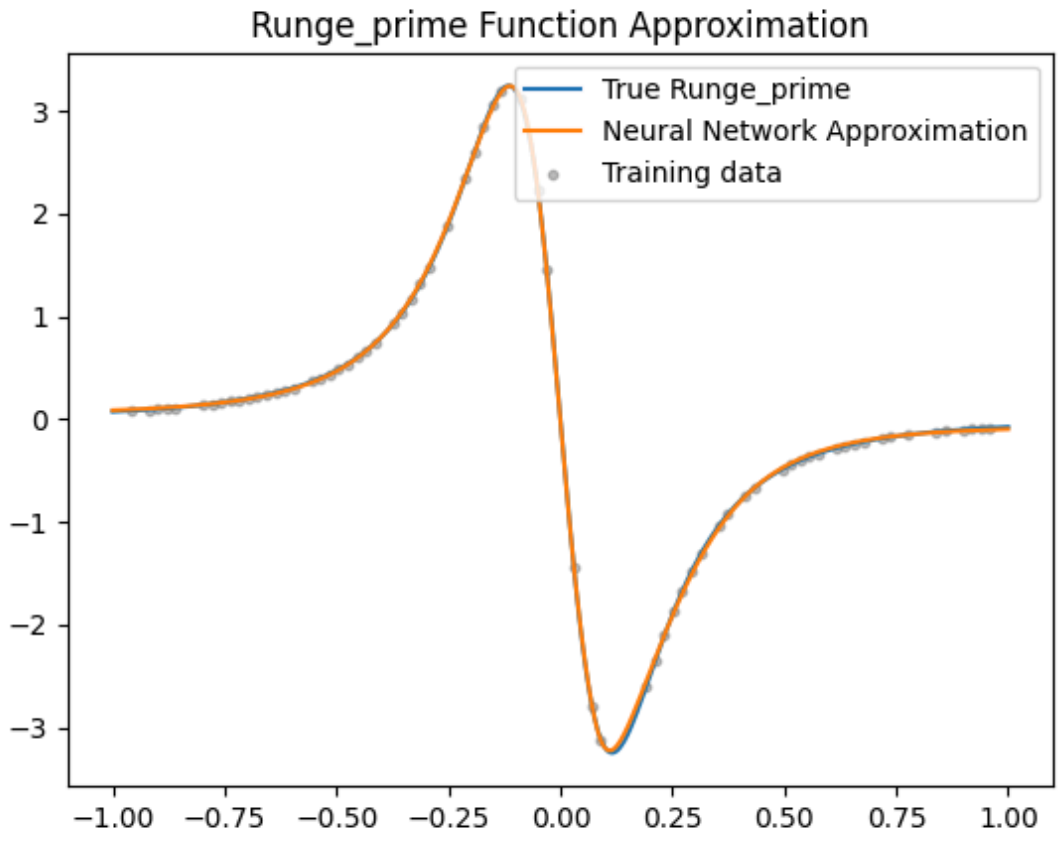

Figure 2. The error between true runge prime and NN approximation.

2

The neural network is used with one input layer with one feature, two hidden layer with 64 neurons in each layer, and one ouput layer with a neuron. The dataset is consisted of 100 data points. 70 percent of them is training set, 15 percent of them is validation set while the remaing 15 percent is testing set. I use $tanh$ as activation funcion, and MSE is loss function to estimate the perform of the approximation, and Adam optimizer to opotimize the parameters.

```
Derivative Train MSE: 1.798259e-02, Max|err|: 4.783448e-01
Derivative Test  MSE: 2.343292e-02, Max|err|: 3.897660e-01
```

Figure 3. The MSE and Absolute Max errors of training set and testing set of f prime

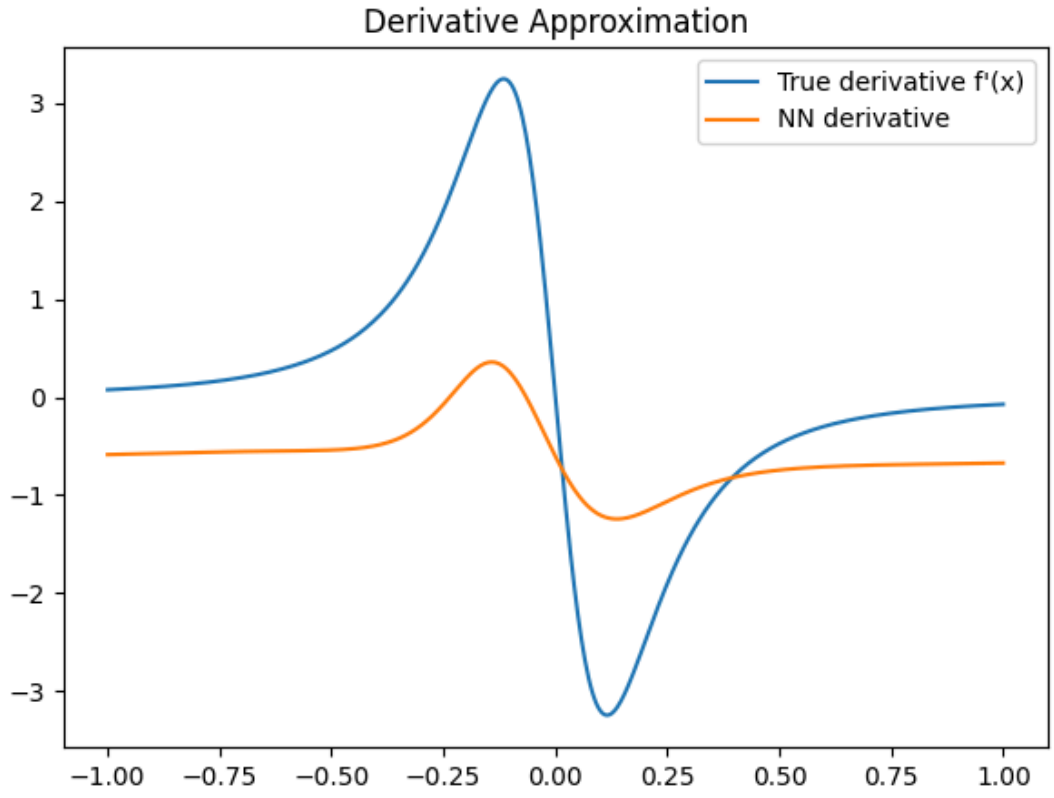

Figure 4. The derivative of the Runge function and the derivative of the neural network

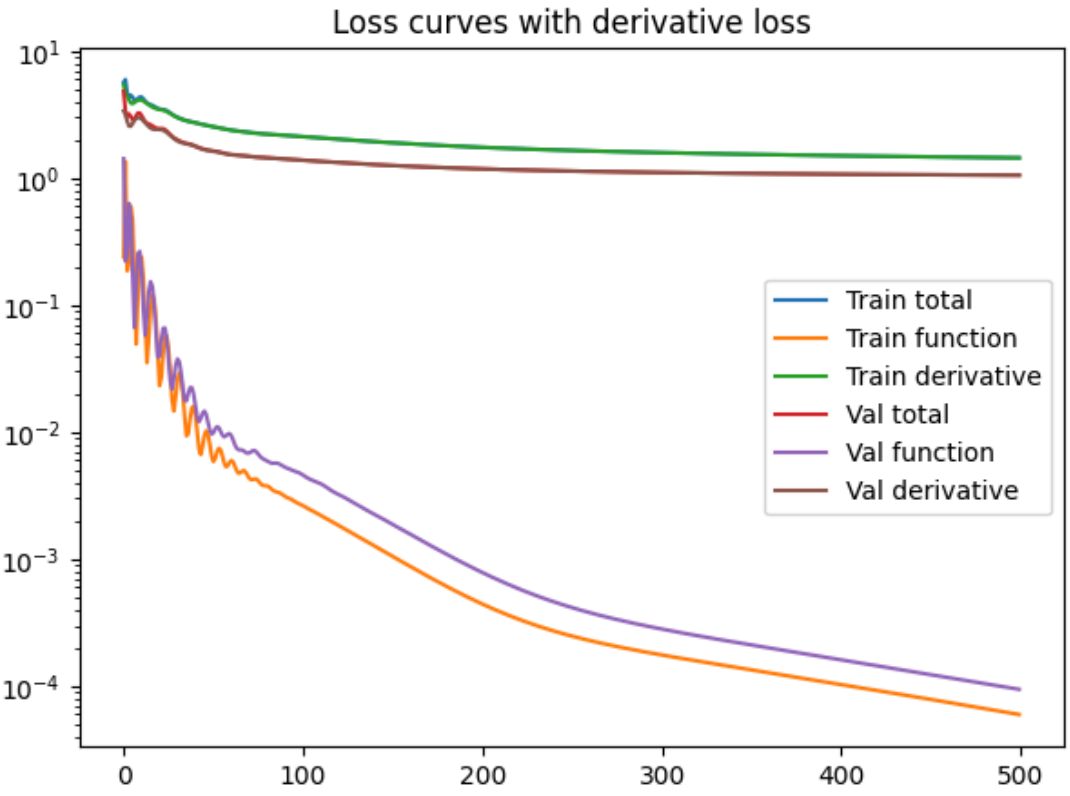

Figure 5. The loss curves of training set and validation set with orginal function and derivative one

The definition of the loss function is not differ from Assignment 2. I still use MSE loss in this Assignment.
I only add

- derivative loss
- calculating function loss, derivative loss and total loss while training
- draw the function/derivative approximation, loss curves
- output the mse and absolute max error of the derivative function

In Figure 2, we can obviously see that the neural network did not approximate the derivative Runge function well.

In Figure 3, we can see the approximation loss grows while derivative the function.

Hence, we might need more neurons or bigger dataset to make the neural network train well, or a better loss function to make the error lower with the same dataset.