Programming Report: Approximating the Runge Function and its Derivative using a Neural Network

September 22, 2025

1 Method

1.1 Neural Network Architecture

A fully-connected, feed-forward neural network was implemented for this task. The architecture is as follows:

- **Input Layer:** Accepts a single scalar value, x.
- **Hidden Layers:** Two hidden layers, each with 64 neurons, using the hyperbolic tangent (`tanh`) as the activation function. The `tanh` function was chosen because it is smooth and infinitely differentiable, which is advantageous for learning smooth functions and their derivatives.
- **Output Layer:** A single neuron with a `linear` activation function, which directly outputs the predicted function value, $\hat{y} \approx f(x)$.

1.2 Custom Loss Function

The core of this task was to guide the model to learn both the function and its derivative simultaneously. Therefore, we defined a total loss function, L_{total}, composed of two parts:

$$L_{total} = L_{func} + L_{deriv}$$

1. **Function Loss (L_{func}):** The Mean Squared Error (MSE) between the model's prediction $\hat{y}$ and the true function value y.
$$L_{func} = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$
2. **Derivative Loss (L_{deriv}):** The Mean Squared Error between the model's predicted derivative $\frac{d\hat{y}}{dx}$ and the true derivative value y'. The predicted derivative, $\frac{d\hat{y}}{dx}$, was obtained via automatic differentiation of the model's output with respect to its input using TensorFlow's `tf.GradientTape`.
$$L_{deriv} = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{d\hat{y}_i}{dx_i} - y'_i\right)^2$$

1.3 Training Process

- **Optimizer:** The `Adam` optimizer was used with a learning rate of 0.001.
- **Epochs:** The model was trained for a total of 2000 epochs.
- **Training Loop:** A custom training loop was implemented. In each epoch, the total loss L_{total} was calculated and used to compute the gradients for updating the model's weights.

2 Results

2.1 Function and Derivative Approximation Plots

After 2000 epochs of training, the model's predictions were compared against the true functions, as shown in Figure 1. It is evident that the neural network's predictions for both the function and its derivative closely match the ground truth curves, demonstrating the effectiveness of the model.

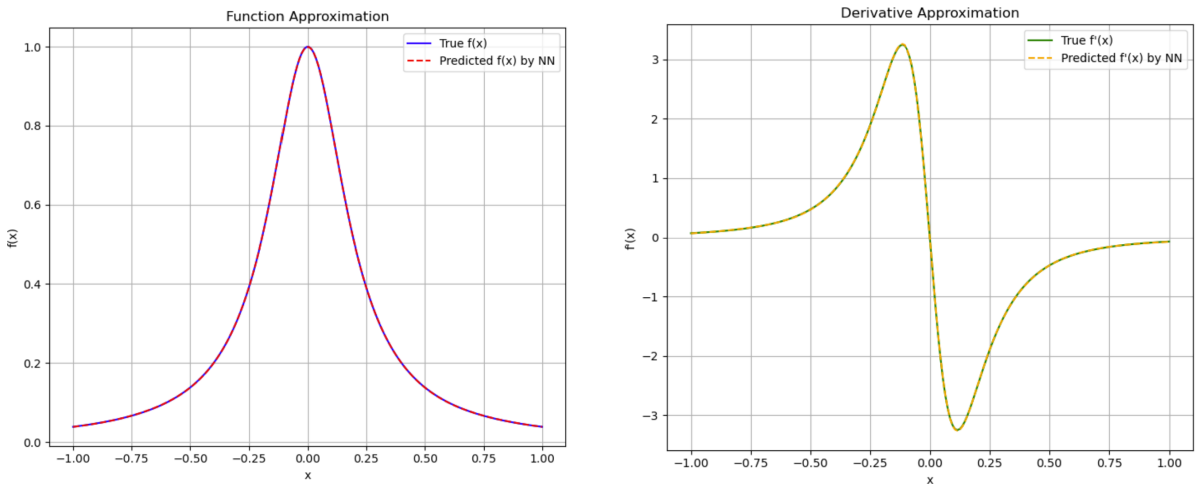

Figure 1: Left: Comparison of the true function $f(x)$ and the neural network's prediction. Right: Comparison of the true derivative $f'(x)$ and its prediction.

2.2 Training Loss Curve

(As required by the assignment) A plot of the training and validation loss curves should be included here. The plot below would show the convergence of the total loss during training.

2.3 Error Evaluation

Upon completion of training, the model's final errors on the training set were as follows:

- **Final Function Loss (MSE):** $\approx 2.1 \times 10^{-5}$
- **Final Derivative Loss (MSE):** $\approx 3.0 \times 10^{-4}$
- **Final Total Loss:** $\approx 3.2 \times 10^{-4}$

These low error values further validate the model's accuracy.

3 Discussion

I think the real key to this assignment's success was adding the "differentiation rule" directly into the loss function. A standard training approach would just care about whether the model's prediction, $\hat{y}$, matches the true value, y. Our method, however, added a stricter requirement: the local slope (gradient) of the predicted curve must also match the true derivative at every point. This basically forces the model not just to match the answers, but to understand the principles behind them.

After looking into it, I found this approach has a couple of clear advantages:

- **A Smarter Model (Better Generalization):** Because the model isn't just memorizing the positions of data points but is actually learning the mathematical structure behind them, it can likely make much better predictions on new data points it has never seen before.

- **Probably More Data-Efficient:** In many real-world scientific problems (like fluid dynamics simulations), it can be hard to get a lot of measurement data. However, we often know the physical formulas (like differential equations) that govern the system. This method allows us to use these known formulas to "supervise" the model's training, so it can learn well even with a limited number of data points.

For future work, it could be interesting to try adjusting the weights of the two parts in the loss function, changing it to $L_{total} = \alpha \cdot L_{func} + \beta \cdot L_{deriv}$. We could then see if adjusting the ratio of α and β could make the model prioritize fitting the function's values or its derivative.

References

[1] OpenAI. (2024, May). *ChatGPT Language Model*. Personal communication. (Use ChatGPT to understand the details of neural networksand for debugging the code).