# Machine Learning HW2

韋詠欣

Due: September 17, 2025

## Written assignment

### Problem 1

We consider a feedforward neural network defined as follows. For input $x \in \mathbb{R}^{n_1}$, the activations are

$$a^{[1]} = x, \tag{1}$$

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}, \quad l = 2, \ldots, L, \tag{2}$$

$$a^{[l]} = \sigma(z^{[l]}). \tag{3}$$

Here $W^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$, $b^{[l]} \in \mathbb{R}^{n_l}$, and $\sigma$ is an activation function applied componentwise. We assume the final layer size is $n_L = 1$, so $a^{[L]}(x)$ is a scalar function of $x$. The goal is to compute

$$\nabla a^{[L]}(x) = \frac{\partial a^{[L]}(x)}{\partial x},$$

which is an $n_1$-dimensional column vector. The algorithm below mirrors backpropagation, and each step includes the reason why it takes this form.

---

**Algorithm 1** Compute $\nabla_x a^{[L]}(x)$ when $n_L = 1$

---

1: **Input:** $\{W^{[l]}, b^{[l]}\}_{l=2}^{L}$, activation $\sigma, \sigma'$, input $x$.

2: **Forward pass (to prepare derivatives):** Set $a^{[1]} \leftarrow x$. For $l = 2, \ldots, L$:

$$z^{[l]} \leftarrow W^{[l]} a^{[l-1]} + b^{[l]}, \quad a^{[l]} \leftarrow \sigma(z^{[l]})$$

Also store $D^{[l]} \leftarrow \mathrm{diag}(\sigma'(z^{[l]}))$. *(We must do this so that $\sigma'(z^{[l]})$ is available for the chain rule.)*

3: **Initialize sensitivity at the output:**

$$u \leftarrow \sigma'(z^{[L]})^\top W^{[L]} \quad (1 \times n_{L-1} \text{ row vector})$$

*(Since $a^{[L]}$ is a scalar, its derivative w.r.t. $a^{[L-1]}$ is a row vector.)*

4: **Backward propagation of the sensitivity:** For $l = L-1, L-2, \ldots, 2$:

$$u \leftarrow u(D^{[l]} W^{[l]})$$

*(Each step applies the chain rule by multiplying the Jacobian $D^{[l]} W^{[l]}$.)*

5: **Output gradient:**

$$\nabla_x a^{[L]}(x) = u^\top \in \mathbb{R}^{n_1}.$$

*(Transpose converts the row vector into the desired column gradient.)*

---

### Problem 2

I don't know what questions to ask.

# Programming assignment

## Method

The dataset consisted of $N = 1000$ points uniformly sampled from $[-1, 1]$, with train/validation/test splits of 70%, 15%, and 15%. The neural network architecture was

$$1 \rightarrow 50 \rightarrow 50 \rightarrow 1,$$

with tanh activation in the hidden layers and a linear output for regression. Training used the Adam optimizer (learning rate 0.01, batch size 64, 200 epochs) and mean squared error (MSE) loss.

## Results

Figure 1 shows the true Runge function and the neural network prediction. The network captures the main shape of the function, but may slightly deviate near the edges.

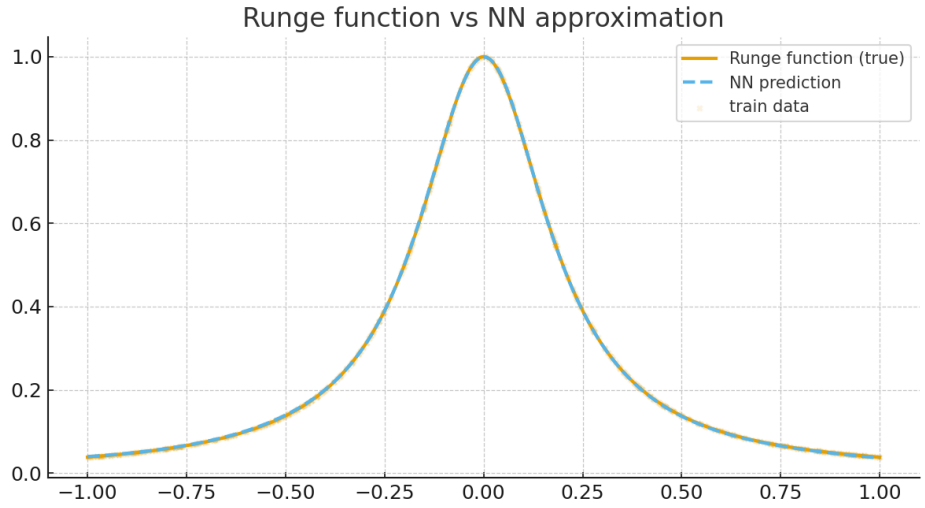


Figure 1: Runge function (solid) and neural network prediction (dashed), with training samples (dots).

The training and validation loss curves are presented in Figure 2. Both losses decrease smoothly and converge to values near $10^{-6}$, with no significant overfitting observed.

Quantitative error results on the test set are given in Table 1.

| Metric | Value |
|---|---|
| Test MSE | $1.39 \times 10^{-6}$ |
| Max absolute error | $2.87 \times 10^{-3}$ |

Table 1: Error metrics for the neural network on the test set.

## Discussion

Neural networks can approximate smooth functions like the Runge function well. Using two hidden layers with enough neurons provides good accuracy. Near the edges of the interval, the network has slightly higher error because the function

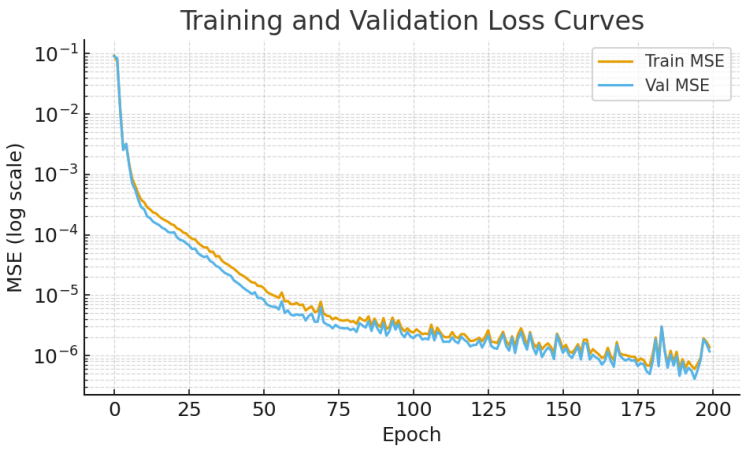


Figure 2: Training and validation MSE versus epochs (log scale).

changes faster there. More training data or deeper networks could improve the approximation.

### Conclusion

We successfully used a simple neural network to approximate the Runge function. The network prediction matches the true function closely, and the errors are small. Neural networks are flexible tools for function approximation.

## References


- chatgpt