HW3

1. Lemma:

Lemma 3.1

Let $k \in \mathbb{N}_0$ and $s \in 2\mathbb{N}-1$. Then it holds that for all $\varepsilon > 0$ there exists a shallow tanh neural network $\Psi_{s,\varepsilon} : [-M, M] \to \mathbb{R}^{\frac{s+1}{2}}$ of width $\frac{s+1}{2}$ such that

$$\max_{\substack{p \le s, \\ p \text{ odd}}} \left\| f_p - (\Psi_{s,\varepsilon})_{\frac{p+1}{2}} \right\|_{W^{k,\infty}} \le \varepsilon, \tag{17}$$

Moreover, the weights of $\Psi_{s,\varepsilon}$ scale as $O\left(\varepsilon^{-s/2}\left(2(s+2)\sqrt{2M}\right)^{s(s+3)}\right)$ for small ε and large s.

Introduction：任意選一個小於S且大於0的奇數p，有一個 neural network $\Psi_{s,\varepsilon} \in \mathbb{R}^{\frac{s+1}{2}}$，其第$\frac{p+1}{2}$個

輸出值會近似一個 monomial function f_p.

Why $\Psi_{s,\varepsilon} : [-M,M] \to \mathbb{R}^{\frac{s+1}{2}}$：

因我們要逼近所有 f_p ($p \le s$ 且 p is odd)，附合條件的p有$\frac{s+1}{2}$個，所

以 $\Psi_{s,\varepsilon}$ 會 output 一個 $\frac{s+1}{2}$-dimension 之向量，且第i項將近似 f_i 之值 $i \in [1, \dots \frac{s+1}{2}]$

Why width $= \frac{p+1}{2}$：

首先 $f_p = y^p$，且p是 odd，f_p是奇函數

而 $\tanh(x) = x - \frac{x^3}{3} + \frac{2x^5}{15} - \frac{17x^7}{315} \cdots = \sum_{n=1}^{\infty} \frac{2^{2n}(2^{2n}-1) B_{2n} x^{2n-1}}{(2n)!}$ where $B_n(n) = \sum_{k=0}^{m} \sum_{v=0}^{k} (-1)^v \binom{k}{v} \frac{(n+v)^m}{k+1}$

By Taylor series, tanh(x) 只含奇數次方的項數 ⇒ 也是奇函數 (*)

為了逼近 f_p，我們至少取 p-th Taylor polynomial of tanh 為訓練的函數，稱其為 ϕ

by (*)，ϕ 只含$\frac{p+1}{2}$個項數，靠調整各項之權重取得近似 f_p 值

所以 hidden layer 要有$\frac{p+1}{2}$個 neurons ⇒ width $= \frac{p+1}{2}$

What is $\max_{\substack{p \le s, \\ p \text{ odd}}} \| f_p - (\Psi_{s,\varepsilon})_{\frac{p+1}{2}} \|_{W^{k,\infty}} \le \varepsilon$：

讓兩個函數在 Sobolev Spaces 上的距離，當任意選擇一個小於S的奇數p，皆小於 ε

The weight of $\Psi_{s,\varepsilon}$ scale as $O(\varepsilon^{-s/2}(2(s+2)\sqrt{2M})^{s(s+3)})$ for small ε and large S：

This shows the cost of approximation: as $\varepsilon \to 0$ (very small error), weights blow up like $\varepsilon^{-s/2}$

As s increases, the blow-up is even more severe.

Lemma 3.2

Let $k \in \mathbb{N}_0$, $s \in 2\mathbb{N}-1$ and $M > 0$. For every $\varepsilon > 0$, there exists a shallow tanh neural network $\psi_{s,\varepsilon} : [-M, M] \to \mathbb{R}^s$ of width $\frac{3(s+1)}{2}$ such that

$$\max_{p \le s} \left\| f_p - \left(\psi_{s,\varepsilon}\right)_p \right\|_{W^{k,\infty}} \le \varepsilon. \tag{26}$$

Furthermore, the weights scale as $O\left(\varepsilon^{-s/2}\left(\sqrt{M}\,(s+2)\right)^{3s(s+3)/2}\right)$ for small ε and large s.

Introduction：任意選一個小於s且大於0的數 p，有一個 neural network $\psi_{s,\varepsilon} \in \mathbb{R}^s$，其第 p 個輸出值會近似一個 monomial function f_p.

Why $\psi_{s,\varepsilon} : [-M, M] \to \mathbb{R}^s$：

因我們要逼近所有 $f_p (p \le s)$，總共 s個，所以 $\psi_{s,\varepsilon}$ 會 output 一個 s-dimension 之向量，且第 i 項將近似 f_i 之值

Why width $= \frac{3(s+1)}{2}$：

若 p 是奇數，在 lemma 3.1 中，可直接用 tanh 逼近（花大約 2 neurons）

而 p-1 是偶數：

因 tanh 的展開式只包含奇數次方，至少取 p -th Taylor polynomial of tanh

我們可用奇函數點對稱性質將最高次的 x^p 消掉，例 tanh(x+a) 和 tanh(x-a)

⇒ 偶數次方的項數留下，而我們訓練這偶函數，使其逼近 f_p

因可重複用奇數的 neural ⇒ 花約 1 neuron

⇒ 所以一對奇偶函數平均花 3 neuron

因 s 為奇數 ⇒ $\underbrace{\frac{s+1}{2}}_{\text{組對奇偶}} \times 3$ → 平均一對奇偶花 3 neuron

Why $\max_{p \le s} \| f_p - (\psi_{s,\varepsilon})_p \|_{W^{k,\infty}} \le \varepsilon$：

表示確定每個 $(\psi_{s,\varepsilon})_p$ 逼近 f_p，在 sobolev space 中，所有 $p \le s$，$(\psi_{s,\varepsilon})_p$ 和 f_p 的距離都小於一開始任意給定的值 ε.

The weight scale as $O(\varepsilon^{-\frac{s}{2}}(\sqrt{M}(s+2))^{3s(s+3)/2})$ for small ε and large s：

當 $\varepsilon \to 0$ 或 $s \to \infty$ 時，權重會增加

2. Program:

(1) Introduction:

Use a neural network to approximate both the Runge function and its derivative.

(2) Method:

(a) Dataset: Random uniform sampling in [-1, 1]. Training: Validation = 80:20.

(b) Model architecture:

a. Numbers of layer: 4

b. Hidden size: [64, 64]

c. Activation: tanh

(c) Training:

a. Loss function:

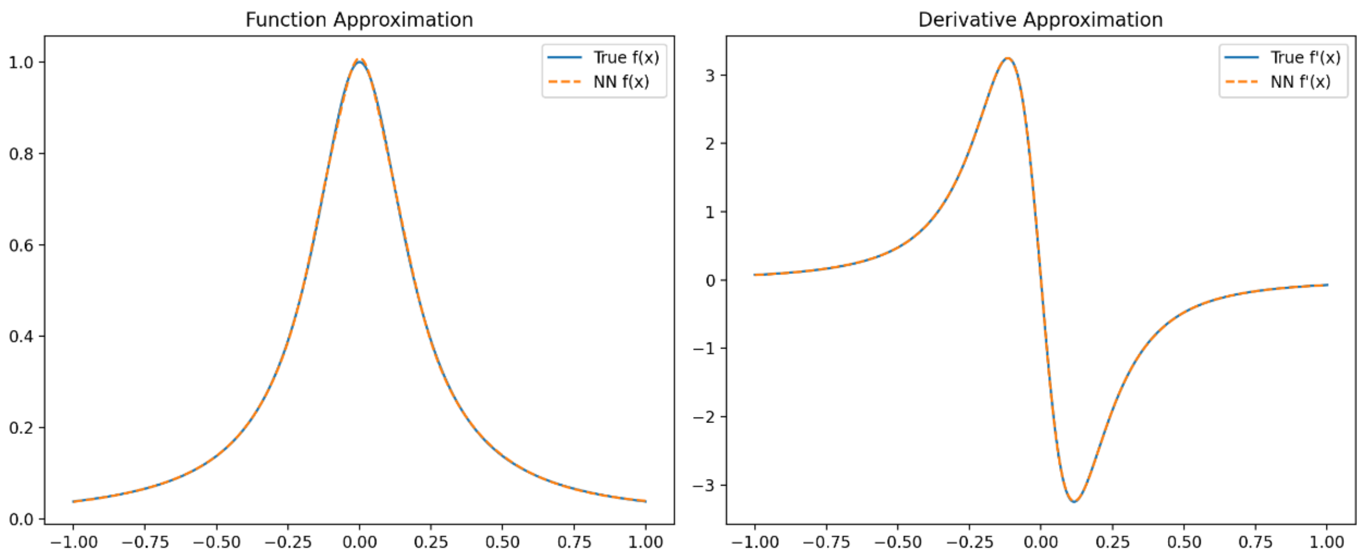
$$\text{loss function}: \frac{1}{N}\sum_{i=1}^{N} \| f_{true}(x_i) - f_{predict}(x_i) \|^2 + \| f'_{true}(x_i) - f'_{predict}(x_i) \|^2$$

b. Optimizer: Adam optimizer

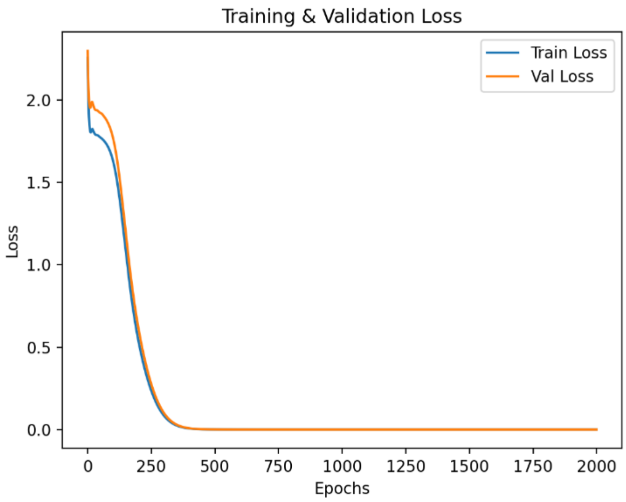
c. Learning rate:1e-3

(3) Result:

(a) The true function and the neural network prediction:



(b) The training/Validation loss curves:



(c) Compute and report errors (MSE or max error):

```
Epoch 1/2000 train MSE=2.254505 val MSE=2.297833
Epoch 200/2000 train MSE=0.541207 val MSE=0.618725
Epoch 400/2000 train MSE=0.006625 val MSE=0.008052
Epoch 600/2000 train MSE=0.000185 val MSE=0.000195
Epoch 800/2000 train MSE=0.000073 val MSE=0.000072
Epoch 1000/2000 train MSE=0.000054 val MSE=0.000054
Epoch 1200/2000 train MSE=0.000044 val MSE=0.000044
Epoch 1400/2000 train MSE=0.000036 val MSE=0.000037
Epoch 1600/2000 train MSE=0.000030 val MSE=0.000031
Epoch 1800/2000 train MSE=0.000024 val MSE=0.000025
Epoch 2000/2000 train MSE=0.000020 val MSE=0.000020

--- Error Metrics ---
Function MSE: 6.752685e-06, Max Error: 0.009458
Derivative MSE: 1.309726e-05, Max Error: 0.013237
```

(4) Discussion:

The result has shown that the neural network has completely approximate both function and its derivative. In the first picture, the prediction function curve is similar to the true function curve, and also there is no obvious difference between the prediction derivative function curve and the true derivative function curve. In the loss curve, since the validation loss closely tracked the training loss, there is no obvious overfitting. In the last picture, MSE=1.309726e-05, indicating that the network achieves a good fit.

(5) Code:

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# --- Runge function and derivative ---
def runge(x):
    return 1.0/(1.0+25.0*x**2)

def Drunge(x):
    return (-50.0*x)/((1.0+25.0*x**2)**2)

# --- Neural network ---
class MLP(nn.Module):
    def __init__(self, hidden_sizes=[64, 64]):
        super().__init__()
        layers = []
        in_dim = 1
        for h in hidden_sizes:
            layers.append(nn.Linear(in_dim, h))
            layers.append(nn.Tanh())
            in_dim = h
        layers.append(nn.Linear(in_dim, 2))
        self.net = nn.Sequential(*layers)
    def forward(self, x):
        return self.net(x)

# --- Data ---
N_total=2000
x_all = np.random.uniform(-1, 1, size=(N_total, 1)).astype(np.float32)
y_all = runge(x_all).astype(np.float32)
y_allD = Drunge(x_all).astype(np.float32)
```

```python

# Split into train/val
x_train, x_val, y_train, y_val, yD_train, yD_val = train_test_split(x_all, y_all, y_allD, test_size=0.2, random_state=1)#訓練集和測試集

# Convert to torch tensors
X_train=torch.from_numpy(x_train)
Y_train=torch.from_numpy(y_train)
YD_train=torch.from_numpy(yD_train)
X_val=torch.from_numpy(x_val)
Y_val=torch.from_numpy(y_val)
YD_val=torch.from_numpy(yD_val)

# --- Model, loss, optimizer ---
lr=1e-3
epochs=2000
model =MLP(hidden_sizes=[64, 64])
criterion=nn.MSELoss()
optimizer=optim.Adam(model.parameters(), lr=lr)
train_losses=[]
val_losses=[]

# --- Training loop ---
for ep in range(epochs):
    model.train()#start training by using training data
    running_loss=0.0
    optimizer.zero_grad()
    out=model(X_train)
    y_pred=out[:, 0]
    yD_pred=out[:, 1]
    y_loss=criterion(Y_train.squeeze(), y_pred)
    yD_loss=criterion(YD_train.squeeze(), yD_pred)
    loss=y_loss+yD_loss
    loss.backward()
    optimizer.step()

    # Validation loss
    model.eval()
    with torch.no_grad():
        out_val=model(X_val)
        f_pred, fD_pred=out_val[:, 0], out_val[:, 1]
        loss_val=criterion(f_pred, Y_val.squeeze())+criterion(fD_pred, YD_val.squeeze())
    train_losses.append(loss.item())
    val_losses.append(loss_val.item())

    if (ep+1)%200 == 0 or ep == 0:
        print(f'Epoch {ep+1}/{epochs} train MSE={loss.item():.6f} val MSE={loss_val.item():.6f}')

# --- Predictions on all points ---
model.eval()
with torch.no_grad():
    out_all = model(torch.from_numpy(x_all))
    f_pred_all = out_all[:,0].numpy()
    df_pred_all = out_all[:,1].numpy()

# --- Error metrics ---
mse_f = mean_squared_error(y_all.squeeze(), f_pred_all)
mse_df = mean_squared_error(y_allD.squeeze(), df_pred_all)
maxerr_f = np.max(np.abs(y_all.squeeze() - f_pred_all))
maxerr_df = np.max(np.abs(y_allD.squeeze() - df_pred_all))

print("\n--- Error Metrics ---")
print(f"Function MSE: {mse_f:.6e}, Max Error: {maxerr_f:.6f}")
print(f"Derivative MSE: {mse_df:.6e}, Max Error: {maxerr_df:.6f}")

# --- Plots ---
#grid
x_grid=np.linspace(-1, 1, 500, dtype=np.float32).reshape(-1, 1)#convert to an array with 1 column
y_true_grid=runge(x_grid)
yD_true_grid=Drunge(x_grid)
model.eval()
```

```python
with torch.no_grad():
    out_grid = model(torch.from_numpy(x_grid))
    f_pred_grid = out_grid[:,0].numpy()
    df_pred_grid = out_grid[:,1].numpy()

plt.figure(figsize=(12,5))

# Function
plt.subplot(1,2,1)
plt.plot(x_grid, y_true_grid, label="True f(x)")
plt.plot(x_grid, f_pred_grid, "--", label="NN f(x)")
plt.legend()
plt.title("Function Approximation")

# Derivative
plt.subplot(1,2,2)
plt.plot(x_grid, yD_true_grid, label="True f'(x)")
plt.plot(x_grid, df_pred_grid, "--", label="NN f'(x)")
plt.legend()
plt.title("Derivative Approximation")

plt.tight_layout()
plt.savefig("function_and_derivative.png", dpi=200)
plt.show()

# Loss curves
plt.figure()
plt.plot(train_losses, label="Train Loss")
plt.plot(val_losses, label="Val Loss")
plt.legend()
plt.title("Training & Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.savefig("loss_curves.png", dpi=200)
plt.show()
```