Machine Learning Assignment 03

313652008 黃睿帆

September 21, 2025

Programming assignment 02

First question

Use the same code from Assignment 2 - programming assignment 01 to calculate the error in approximating the derivative of the given function.Recall that in programming assignment 1, we use *tanh* to apporximate the given function, by supervised learning (neural network):

1. Hypothsis function: $\hat{f} = tanh$
2. Hidden layer: 2
3. neurons in each layer: 50
4. activation function: *tanh*
5. loss function: (MSE) $L(\theta) = \frac{1}{N}\sum_{i=1}^{N}(\hat{f}'(x_i;\theta) - f'(x_i))^2$

Here we using AI tools again and the same function (NOTE: Some of the following Python codes are generated by chat-GPT, and the Python codes were runned by Colab), to acheive our goal:

1. Plotting the true function and the neural network prediction together.
2. Showing the training/validation loss curves.
3. Computing and report errors (MSE or max error).

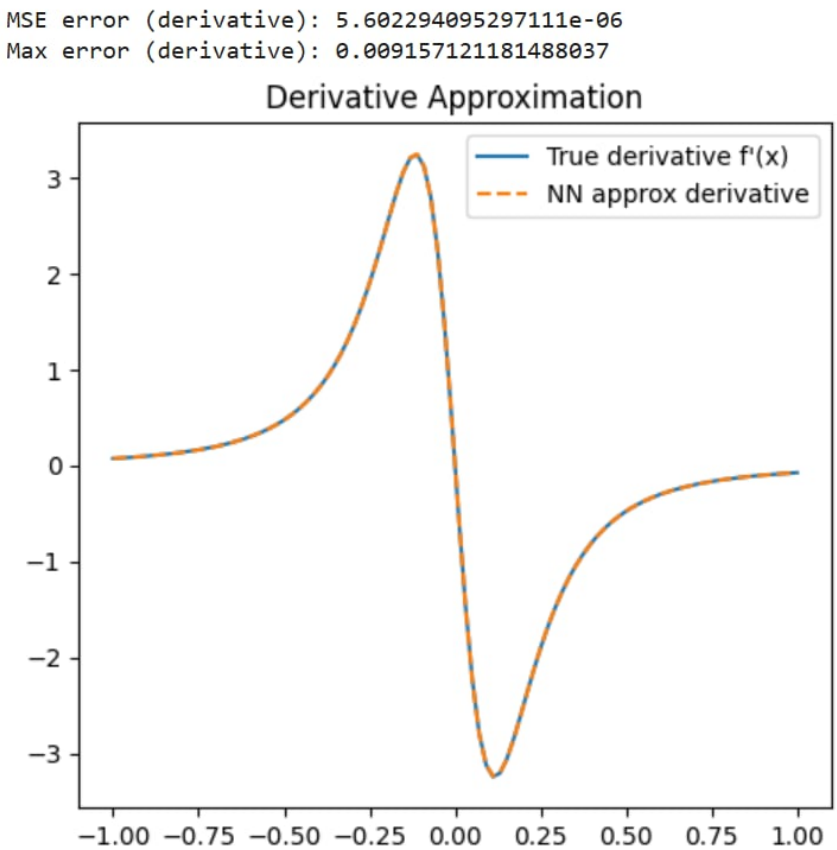
The Python code:

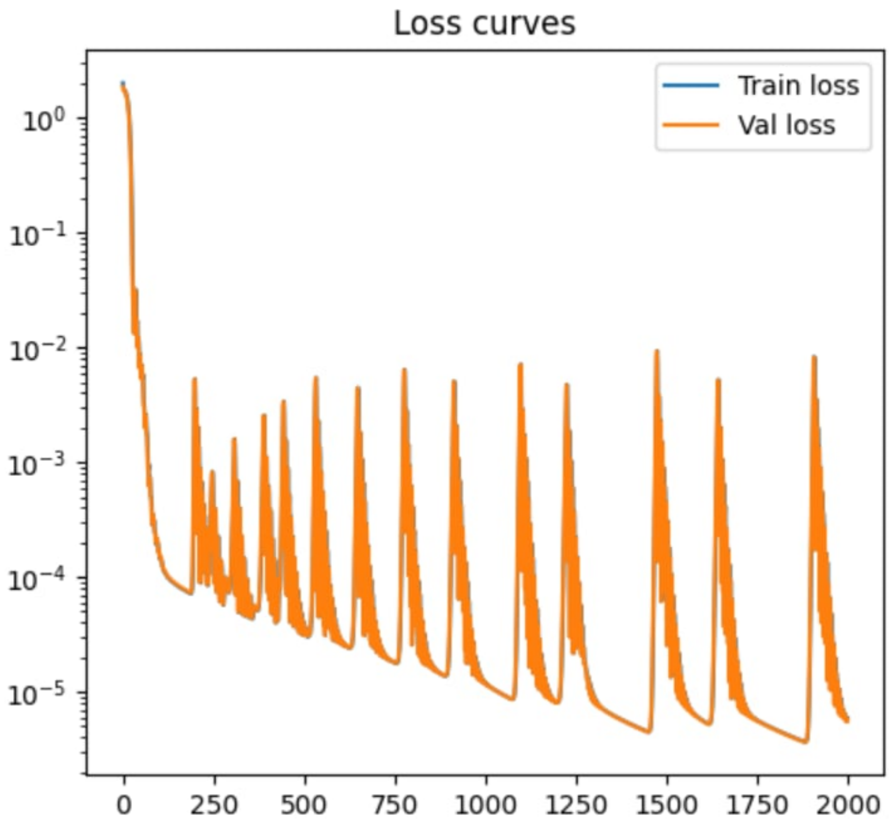
```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt

# Runge function and its derivative
def runge(x):
    return 1 / (1 + 25 * x**2)

def runge_derivative(x):
    return -50 * x / (1 + 25 * x**2)**2

# Training data
np.random.seed(0)
x_train = np.linspace(-1, 1, 200).reshape(-1, 1)
y_train = runge_derivative(x_train)

x_val = np.linspace(-1, 1, 100).reshape(-1, 1)
y_val = runge_derivative(x_val)

x_train_torch = torch.tensor(x_train, dtype=torch.float32, requires_grad=True)
y_train_torch = torch.tensor(y_train, dtype=torch.float32)

x_val_torch = torch.tensor(x_val, dtype=torch.float32, requires_grad=True)
y_val_torch = torch.tensor(y_val, dtype=torch.float32)

# Neural network definition
```

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.hidden1 = nn.Linear(1, 50)
        self.hidden2 = nn.Linear(50, 50)
        self.out = nn.Linear(50, 1)
        self.tanh = nn.Tanh()

    def forward(self, x):
        x = self.tanh(self.hidden1(x))
        x = self.tanh(self.hidden2(x))
        x = self.out(x)
        return x

# Instantiate model, optimizer, loss
model = Net()
optimizer = optim.Adam(model.parameters(), lr=0.01)
criterion = nn.MSELoss()

train_losses, val_losses = [], []

# Training loop
epochs = 2000
for epoch in range(epochs):
    # Compute NN output for training
    y_pred_train = model(x_train_torch)

    # Compute NN derivative wrt input (autograd)
    dydx_train = torch.autograd.grad(
        outputs=y_pred_train,
        inputs=x_train_torch,
        grad_outputs=torch.ones_like(y_pred_train),
        create_graph=True
    )[0]

    loss = criterion(dydx_train, y_train_torch)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # Validation
    y_pred_val = model(x_val_torch)
    dydx_val = torch.autograd.grad(
        outputs=y_pred_val,
        inputs=x_val_torch,
        grad_outputs=torch.ones_like(y_pred_val),
        create_graph=True
    )[0]
    val_loss = criterion(dydx_val, y_val_torch)

    train_losses.append(loss.item())
    val_losses.append(val_loss.item())

# Evaluate performance
y_pred = model(x_val_torch)
dydx_pred = torch.autograd.grad(
    outputs=y_pred,
    inputs=x_val_torch,
    grad_outputs=torch.ones_like(y_pred),
    create_graph=True
)[0]

mse_error = criterion(dydx_pred, y_val_torch).item()
max_error = torch.max(torch.abs(dydx_pred - y_val_torch)).item()

print("MSE error (derivative):", mse_error)
print("Max error (derivative):", max_error)

# Plot true vs predicted derivative
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(x_val, y_val, label="True derivative f'(x)")
```

```
plt.plot(x_val, dydx_pred.detach().numpy(), '--', label="NN approx derivative")
plt.legend()
plt.title("Derivative Approximation")

# Plot training/validation loss curves
plt.subplot(1,2,2)
plt.plot(train_losses, label="Train loss")
plt.plot(val_losses, label="Val loss")
plt.yscale("log")
plt.legend()
plt.title("Loss curves")
plt.show()
```

The results are as follows:





Second question

Use a neural network to approximate both the Runge function and its derivative. i.e. Training a neural network that approximates: The function $f(x)$ and its derivative $f'(x)$.

The Python code:

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt

# --- Define Runge function and derivative ---
def runge(x):
    return 1 / (1 + 25 * x**2)

def runge_derivative(x):
    return -50 * x / (1 + 25 * x**2)**2

# --- Training / Validation data ---
np.random.seed(0)
x_train = np.linspace(-1, 1, 200).reshape(-1, 1)
y_train = runge(x_train)
dy_train = runge_derivative(x_train)

x_val = np.linspace(-1, 1, 100).reshape(-1, 1)
y_val = runge(x_val)
dy_val = runge_derivative(x_val)

x_train_torch = torch.tensor(x_train, dtype=torch.float32, requires_grad=True)
y_train_torch = torch.tensor(y_train, dtype=torch.float32)
dy_train_torch = torch.tensor(dy_train, dtype=torch.float32)

x_val_torch = torch.tensor(x_val, dtype=torch.float32, requires_grad=True)
y_val_torch = torch.tensor(y_val, dtype=torch.float32)
```

```
dy_val_torch = torch.tensor(dy_val, dtype=torch.float32)

# --- Neural network definition ---
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.hidden1 = nn.Linear(1, 50)
        self.hidden2 = nn.Linear(50, 50)
        self.out = nn.Linear(50, 1)
        self.tanh = nn.Tanh()

    def forward(self, x):
        x = self.tanh(self.hidden1(x))
        x = self.tanh(self.hidden2(x))
        x = self.out(x)
        return x

model = Net()
optimizer = optim.Adam(model.parameters(), lr=0.01)
mse_loss = nn.MSELoss()

# --- Training loop ---
epochs = 2000
train_losses, val_losses = [], []

for epoch in range(epochs):
    # Training forward pass
    y_pred_train = model(x_train_torch)

    # Compute derivative via autograd
    dy_pred_train = torch.autograd.grad(
        outputs=y_pred_train,
        inputs=x_train_torch,
        grad_outputs=torch.ones_like(y_pred_train),
        create_graph=True
    )[0]

    # Loss = function loss + derivative loss
    loss_func = mse_loss(y_pred_train, y_train_torch)
    loss_deriv = mse_loss(dy_pred_train, dy_train_torch)
    loss = loss_func + loss_deriv

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # Validation
    y_pred_val = model(x_val_torch)
    dy_pred_val = torch.autograd.grad(
        outputs=y_pred_val,
        inputs=x_val_torch,
        grad_outputs=torch.ones_like(y_pred_val),
        create_graph=True
    )[0]

    val_loss = mse_loss(y_pred_val, y_val_torch) + mse_loss(dy_pred_val, dy_val_torch)

    train_losses.append(loss.item())
    val_losses.append(val_loss.item())

# --- Final predictions ---
x_plot = np.linspace(-1, 1, 500).reshape(-1, 1)
y_true = runge(x_plot)
dy_true = runge_derivative(x_plot)

x_plot_torch = torch.tensor(x_plot, dtype=torch.float32, requires_grad=True)

#
y_pred = model(x_plot_torch)
dy_pred = torch.autograd.grad(
    outputs=y_pred,
    inputs=x_plot_torch,
    grad_outputs=torch.ones_like(y_pred),
```

```
    create_graph=True
)[0]

# detach only when converting to numpy
y_pred_np = y_pred.detach().numpy().flatten()
dy_pred_np = dy_pred.detach().numpy().flatten()


# --- Compute errors ---
mse_f = np.mean((y_pred_np - y_true.flatten())**2)
max_f = np.max(np.abs(y_pred_np - y_true.flatten()))

mse_df = np.mean((dy_pred_np - dy_true.flatten())**2)
max_df = np.max(np.abs(dy_pred_np - dy_true.flatten()))

print("Function Approximation Errors:")
print(f"  MSE: {mse_f:.6f}, Max Error: {max_f:.6f}")
print("Derivative Approximation Errors:")
print(f"  MSE: {mse_df:.6f}, Max Error: {max_df:.6f}")

# --- Plot function approximation ---
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(x_plot, y_true, label="True f(x)")
plt.plot(x_plot, y_pred_np, '--', label="NN f(x)")
plt.legend()
plt.title("Function Approximation")

# --- Plot derivative approximation ---
plt.subplot(1,2,2)
plt.plot(x_plot, dy_true, label="True f'(x)")
plt.plot(x_plot, dy_pred_np, '--', label="NN f'(x)")
plt.legend()
plt.title("Derivative Approximation")
plt.show()

# --- Plot training/validation loss curves ---
plt.figure(figsize=(8,5))
plt.plot(train_losses, label="Train loss")
plt.plot(val_losses, label="Val loss")
plt.yscale("log")
plt.legend()
plt.title("Training and Validation Loss (Function + Derivative)")
plt.show()
```

We use $tanh$ again to apporximate the given function $f(x)$, by supervised learning (neural network):

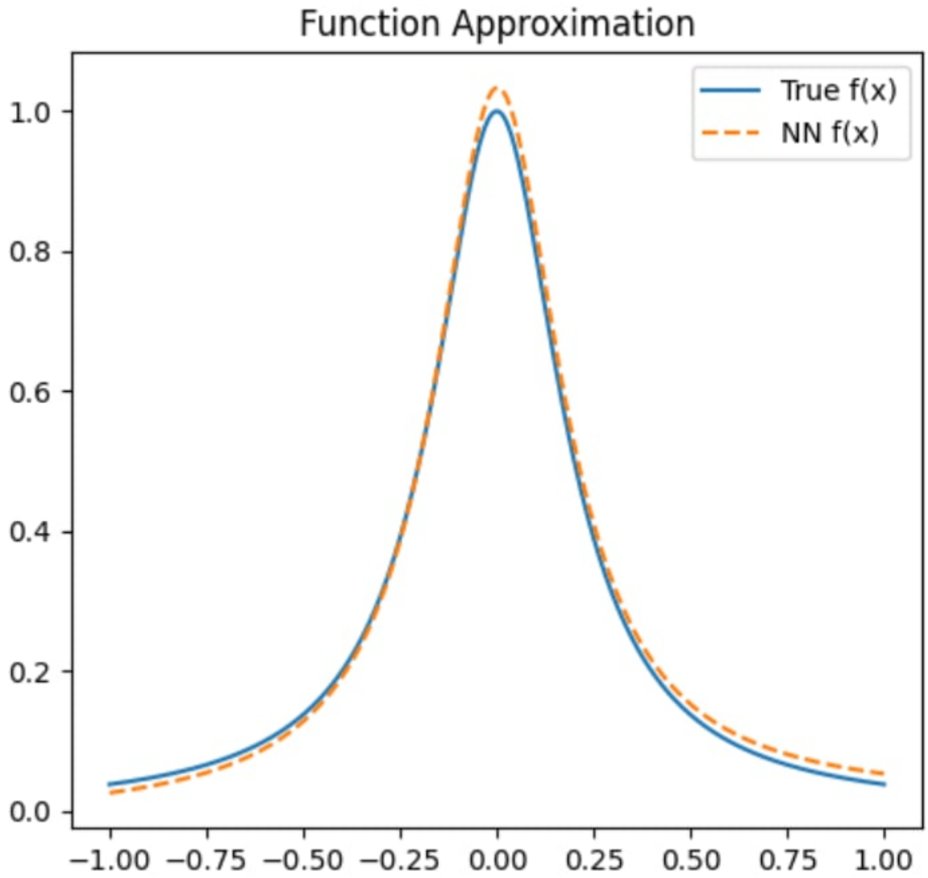
1. Hypothsis function: $\hat{f} = tanh$
2. Hidden layer: 2
3. neurons in each layer: 50
4. activation function: $tanh$
5. loss function: $MSE(\hat{f}, f) + MSE(\hat{f}', f')$

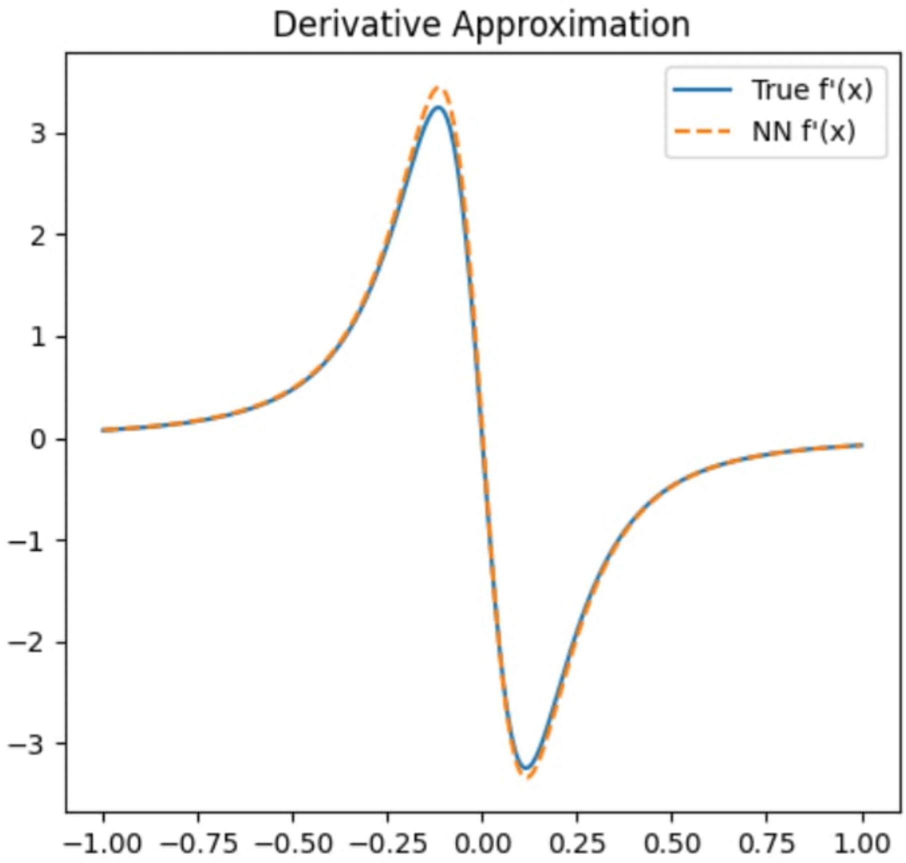
Here we using AI tools again and the same function (NOTE: Some of the following Python codes are generated by chat-GPT, and the Python codes were runned by Colab), to acheive our goal:

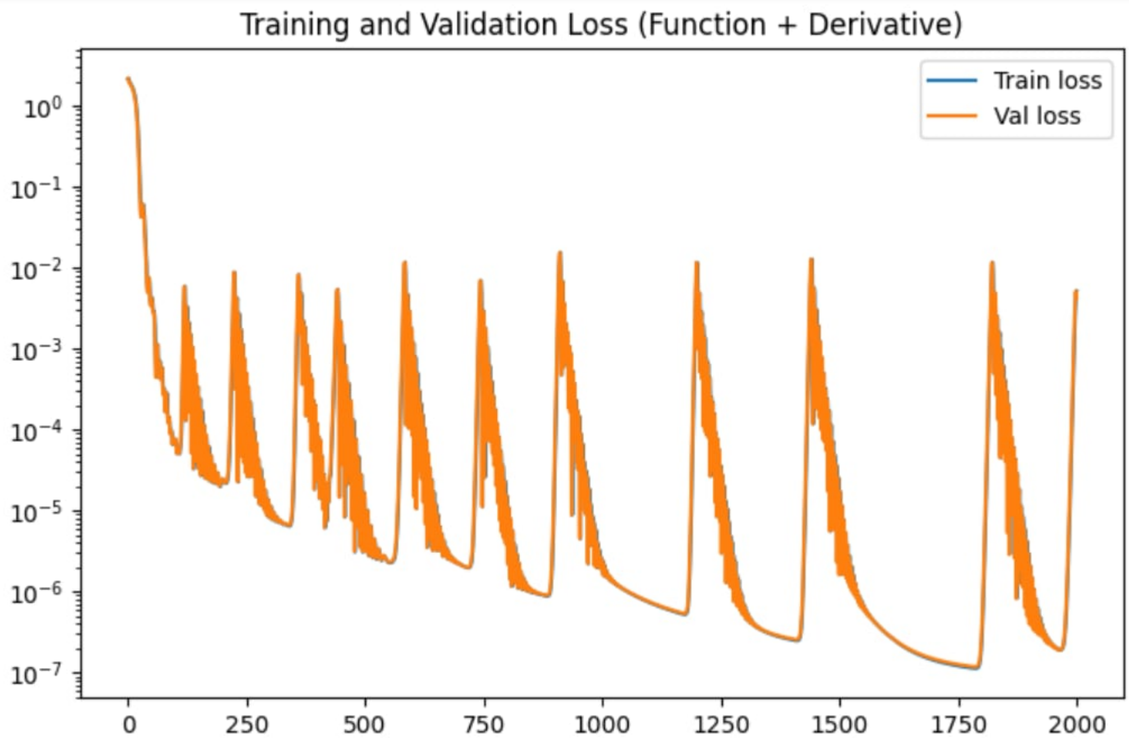
1. Plotting the true function and the neural network prediction together.
2. Showing the training/validation loss curves.
3. Computing and report errors (MSE or max error).

The results are as follows:

```
Function Approximation Errors:
  MSE: 0.000279, Max Error: 0.035552
Derivative Approximation Errors:
  MSE: 0.003994, Max Error: 0.216729
```




Derivative Approximation
True f'(x)
NN f'(x)
3
2
1
0
−1
−2
−3
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00


Training and Validation Loss (Function + Derivative)
Train loss
Val loss
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
0
250
500
750
1000
1250
1500
1750
2000