CEG5306 Homework 3

Zhao Lixiuqi

A0239866J

Qn a:  
code for RBFN with exact interpolation:

import torch

import numpy as np

import matplotlib.pyplot as plt

from sklearn.metrics import mean\_squared\_error

torch.set\_default\_dtype(torch.float64)

def true\_func\_t(x: torch.Tensor) -> torch.Tensor:

return 1.2 \* torch.sin(torch.pi \* x) - torch.cos(2.4 \* torch.pi \* x)

def make\_data\_t(

x\_min=-1.0,

x\_max=1.0,

step\_train=0.05,

step\_test=0.01,

noise\_std=0.3,

seed=7,

):

rng = np.random.default\_rng(seed)

x\_train = torch.tensor(np.arange(x\_min, x\_max + 1e-12, step\_train))

y\_clean\_train = true\_func\_t(x\_train)

noise = torch.tensor(rng.normal(0.0, 1.0, size=x\_train.shape))

y\_train = y\_clean\_train + noise\_std \* noise

x\_test = torch.tensor(np.arange(x\_min, x\_max + 1e-12, step\_test))

y\_test = true\_func\_t(x\_test) *# clean for fair evaluation*

return x\_train, y\_train, x\_test, y\_test

def rbf\_gaussian\_t(x: torch.Tensor, c: torch.Tensor, sigma: float) -> torch.Tensor:

d2 = (x[:, None] - c[None, :]) \*\* 2

return torch.exp(-d2 / (2.0 \* sigma\*\*2))

def solve\_exact\_interpolation\_t(

x\_train: torch.Tensor,

y\_train: torch.Tensor,

sigma: float,

ridge: float = 0.0,

):

centers = x\_train.clone()

Phi = rbf\_gaussian\_t(x\_train, centers, sigma)

if ridge > 0:

G = Phi.T @ Phi + ridge \* torch.eye(Phi.shape[1], dtype=Phi.dtype)

b = Phi.T @ y\_train

w = torch.linalg.solve(G, b)

else:

w = torch.linalg.solve(Phi, y\_train)

return centers, w

def predict\_t(x: torch.Tensor, centers: torch.Tensor, w: torch.Tensor, sigma: float):

Phi = rbf\_gaussian\_t(x, centers, sigma)

return Phi @ w

def rmse(y\_true\_np: np.ndarray, y\_pred\_np: np.ndarray) -> float:

return float(np.sqrt(mean\_squared\_error(y\_true\_np, y\_pred\_np)))

def main():

x\_tr, y\_tr, x\_te, y\_te = make\_data\_t()

sigma = 0.1 *# given*

centers, w = solve\_exact\_interpolation\_t(

x\_tr, y\_tr, sigma=sigma, ridge=0.0)

yhat\_tr = predict\_t(x\_tr, centers, w, sigma)

yhat\_te = predict\_t(x\_te, centers, w, sigma)

mse\_tr = mean\_squared\_error(y\_tr.numpy(), yhat\_tr.detach().numpy())

mse\_te = mean\_squared\_error(y\_te.numpy(), yhat\_te.detach().numpy())

print("Q1(a) PyTorch - Exact Interpolation (σ = 0.1)")

print(f" Train MSE (noisy): {mse\_tr:.6f}")

print(f" Test MSE (clean): {mse\_te:.6f}")

print(f" #centers: {len(centers)}")

xx = x\_te.numpy()

yy\_true = y\_te.numpy()

plt.figure(figsize=(10, 6))

plt.plot(xx, yy\_true, "k-", lw=2, label="True function (clean)")

plt.scatter(x\_tr.numpy(), y\_tr.numpy(), s=18, color="#8888ff",

label="Train (noisy)")

plt.plot(xx, yhat\_te.detach().numpy(), "r-", lw=2,

label="RBFN exact interpolation"

)

plt.text(0.02, 0.98, f'MSE = {mse\_te:.4f}',

transform=plt.gca().transAxes,

fontsize=12,

verticalalignment='top',

bbox=dict(boxstyle='round,pad=0.3', facecolor='white', edgecolor='black'))

plt.title("Q1(a): RBFN Exact Interpolation (Gaussian σ=0.1)")

plt.xlabel("x")

plt.ylabel("y")

plt.legend()

plt.grid(True, alpha=0.3)

plt.tight\_layout()

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

main()

plot obtained (MSE = 0.1007) :  
A graph with red and blue lines

AI-generated content may be incorrect.

As we can see from the result, the RBFN that uses exact interpolation:  
1. Passes through all the noisy points generated during the training;

2. Is not able to generalise well to fit the true function due to the noisy training set.

Qn b:  
code for RBFN with the strategy of “Fixed Centers Selected at Random”:

import torch

import numpy as np

import matplotlib.pyplot as plt

from sklearn.metrics import mean\_squared\_error

torch.set\_default\_dtype(torch.float64)

def true\_func\_t(x: torch.Tensor) -> torch.Tensor:

return 1.2 \* torch.sin(torch.pi \* x) - torch.cos(2.4 \* torch.pi \* x)

def make\_data\_t(

x\_min=-1.0,

x\_max=1.0,

step\_train=0.05,

step\_test=0.01,

noise\_std=0.3,

seed=7,

):

rng = np.random.default\_rng(seed)

x\_train = torch.tensor(

np.arange(x\_min, x\_max + 1e-12, step\_train)

)

y\_clean\_train = true\_func\_t(x\_train)

noise = torch.tensor(rng.normal(0.0, 1.0, size=x\_train.shape))

y\_train = y\_clean\_train + noise\_std \* noise

x\_test = torch.tensor(

np.arange(x\_min, x\_max + 1e-12, step\_test)

)

y\_test = true\_func\_t(x\_test)

return x\_train, y\_train, x\_test, y\_test

def rbf\_gaussian\_t(

x: torch.Tensor, c: torch.Tensor, sigma: float

) -> torch.Tensor:

d2 = (x[:, None] - c[None, :]) \*\* 2

return torch.exp(-d2 / (2.0 \* sigma\*\*2))

def solve\_fixed\_random\_centers\_t(

x\_train: torch.Tensor,

y\_train: torch.Tensor,

num\_centers: int = 20,

ridge: float = 0.0,

seed: int = 42,

):

rng = np.random.default\_rng(seed)

idx = np.sort(rng.choice(len(x\_train), num\_centers, replace=False))

centers = x\_train[idx]

*# Calculate sigma using the formula: σ\_i = d\_max / sqrt(2M)*

*# where d\_max is the maximum distance between chosen centers*

centers\_np = centers.numpy()

distances = []

for i in range(len(centers\_np)):

for j in range(i + 1, len(centers\_np)):

distances.append(abs(centers\_np[i] - centers\_np[j]))

d\_max = max(distances)

sigma = d\_max / np.sqrt(2 \* num\_centers)

Phi = rbf\_gaussian\_t(x\_train, centers, sigma)

G = Phi.T @ Phi + ridge \* torch.eye(Phi.shape[1], dtype=Phi.dtype)

b = Phi.T @ y\_train

w = torch.linalg.solve(G, b)

return centers, w, sigma

def predict\_t(

x: torch.Tensor, centers: torch.Tensor, w: torch.Tensor, sigma: float

) -> torch.Tensor:

Phi = rbf\_gaussian\_t(x, centers, sigma)

return Phi @ w

def main():

x\_tr, y\_tr, x\_te, y\_te = make\_data\_t()

num\_centers = 20

ridge = 0.0

centers, w, sigma = solve\_fixed\_random\_centers\_t(

x\_train=x\_tr,

y\_train=y\_tr,

num\_centers=num\_centers,

ridge=ridge,

seed=42,

)

yhat\_tr = predict\_t(x\_tr, centers, w, sigma)

yhat\_te = predict\_t(x\_te, centers, w, sigma)

mse\_tr = mean\_squared\_error(y\_tr.numpy(), yhat\_tr.detach().numpy())

mse\_te = mean\_squared\_error(y\_te.numpy(), yhat\_te.detach().numpy())

print("Q1(b) Fixed Random Centers — PyTorch")

print(f" #centers: {num\_centers}")

print(f" Train MSE (noisy): {mse\_tr:.6f}")

print(f" Test MSE (clean): {mse\_te:.6f}")

xx = x\_te.numpy()

yy\_true = y\_te.numpy()

yhat\_te\_np = yhat\_te.detach().numpy()

plt.figure(figsize=(10, 6))

plt.plot(xx, yy\_true, "k-", lw=2, label="True function (clean)")

plt.scatter(

x\_tr.numpy(), y\_tr.numpy(), s=18, color="#8888ff",

label="Train (noisy)"

)

plt.plot(

xx, yhat\_te\_np, "g-", lw=2,

label=f"RBFN (20 random centers)"

)

for c in centers.numpy():

plt.axvline(

c, ymin=0.02, ymax=0.98,

color="g", linestyle="dotted", alpha=0.3

)

plt.text(

0.02, 0.98,

f"MSE = {mse\_te:.4f}",

transform=plt.gca().transAxes,

fontsize=12,

verticalalignment="top",

bbox=dict(

boxstyle="round,pad=0.3",

facecolor="white",

edgecolor="black"

)

)

plt.title(f"Q1(b): Fixed Random Centers (20), Gaussian σ={sigma:.4f}")

plt.xlabel("x")

plt.ylabel("y")

plt.legend()

plt.grid(True, alpha=0.3)

plt.tight\_layout()

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

main()

plot obtained (MSE = 0.0375):  
A graph with green and blue lines

AI-generated content may be incorrect.  
The MSE (mean squared error) of the fixed random center approach is much smaller than that of the exact interpolation. (0.0375 < 0.1007)  
  
As we can observe intuitively from the graph, the fixed random center approach generates a result graph that is much more smooth (less oscillations) graph as compared to that of the exact interpolation. This is due to the decrease in the number of neurons in the hidden layer and also a different approach of training. This shows that this approach is better in generalisation such that the characteristic of the true function is better approximated, instead of getting over-fitted result due to noisy training set.

Declaration of use of AI:  
I used GPT5 to help me with the use of pytorch and matplotlib library