

Assignment 3.

✍ Written assignment

1. Reading and Explaining Lemmas

Your task is to read the following paper:

Ryck et al., *On the approximation of functions by tanh neural networks*

[Link to paper](#)

Focus on Lemma 3.1 and Lemma 3.2.

Goal : To claim that neural networks with tanh activation function can approximate polynomials very well (both value and their derivative).

Lemma 3.1 (for odd powers)

For odd p , $p \leq s$, there exists a one-hidden-layer tanh network of width $\frac{s+1}{2}$ that can simultaneously approximate all y^p on the interval $[-M, M]$ with error at most ϵ .

Idea: (1) Expand $\tanh(y)$ in Taylor series

(2) Apply central difference operator S_h^p to eliminate lower-order terms and remain y^p (odd-power)

(3) When h is small enough, the error can be controlled.

Lemma 3.2 (extending to even powers) .

For odd p , there exists a one-hidden-layer tanh network of width $\frac{3(s+1)}{2}$ that can simultaneously approximate all $\{ty^k\}_{k \in N, k \leq s}$ on the interval $[-M, M]$ with error at most ϵ .

Idea: (1) Odd powers are handled by Lemma 3.1

(2) For even power, use algebraic identity:

$$y^{2n} = \frac{1}{2\alpha(2n+1)} \left[(y+\alpha)^{2n+1} - (y-\alpha)^{2n+1} - 2 \sum_{k=0}^{n-1} \binom{2n+1}{2k} \alpha^{2(n-k)+1} y^{2k} \right].$$

\Rightarrow even powers can be expressed in terms of odd powers and lower even powers.

(3) By selecting α , the error can be controlled.

Remark. Lemma 3.2 construct a shallow (one-hidden-layer) network to approximate every monomial up to degree s , so we can approximate any polynomial and then approximate smooth functions.

2. Unanswered Questions

There are unanswered questions from the lecture, and there are likely more questions we haven't covered.

Is there a clear trade-off between network depth versus width in achieving approximation rates?

(Under the constraint of fixed parameter count or computational budget, does there exist a relation that characterizes the optimal depth/width configuration required to minimize the error?)

Programming assignment

1. Use the **same code** from Assignment 2 - programming assignment 1 to calculate the error in approximating the derivative of the given function.

```
#derivative MSE
net.eval()
x_test = np.linspace(-1, 1, 500).reshape(-1, 1).astype(np.float32)
x_test_tensor = torch.from_numpy(x_test)
x_test_tensor.requires_grad_(True)

y_hat = net(x_test_tensor)
dy_dx_hat = torch.autograd.grad(
    outputs = y_hat,
    inputs = x_test_tensor,
    grad_outputs = torch.ones_like(y_hat),
    retain_graph = False,
    create_graph = False,
)[0]

#f'(x) = -50x/(1+25x^2)^2
with torch.no_grad():
    x_np = x_test
    dy_dx_true_np = -(50.0 * x_np) / (1.0 + 25.0 * x_np**2)**2
    dy_dx_true =
        torch.from_numpy(dy_dx_true_np.astype(np.float32))
deriv_mse = torch.mean((dy_dx_hat - dy_dx_true)**2).item()
print(f"Derivative Approximation MSE: {deriv_mse:.6f}")
```

Result: Derivative Approximation MSE : 0.000218 .

2. In this assignment, you will use a neural network to approximate both the **Runge function** and its **derivative**. Your task is to train a neural network that approximates:
- The function $f(x)$ itself.
 - The derivative $f'(x)$.

You should define a **loss function** consisting of two components:

- Function loss:** the error between the predicted $f(x)$ and the true $f(x)$.
- Derivative loss:** the error between the predicted $f'(x)$ and the true $f'(x)$.

Write a short report (1-2 pages) explaining method, results, and discussion including

- Plot the true function and the neural network prediction together.
- Show the training/validation loss curves.
- Compute and report errors (MSE or max error).

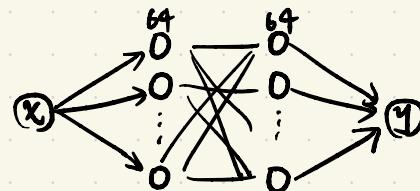
Runge function: $f(x) = \frac{1}{1+25x^2}, x \in [-1, 1]$,

$$f'(x) = \frac{-50x}{(1+25x^2)^2}$$

Method: feedforward neural network

dataset: uniformly sampling 200 points from $[-1, 1]$ and the data was split into 80% training and 20% validation sets.

network architecture:



using tanh activations.

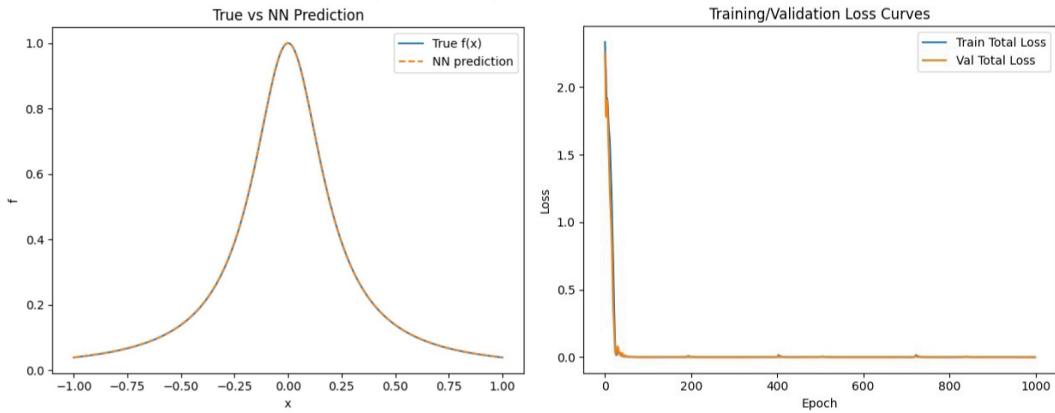
loss function: combined MSE , $\lambda=1$.

$$L := \text{MSE}(f(x), \hat{f}(x)) + \lambda \cdot \text{MSE}(f'(x), \hat{f}'(x))$$

optimizer : Adam , learning rate 0.01 , 1000 epochs

Results :

```
Epoch [200/1000] Train Total: 0.000110 (f: 0.000009, f': 0.000102) | Val Total: 0.001828 (f: 0.000045, f': 0.001783)
Epoch [400/1000] Train Total: 0.001762 (f: 0.000117, f': 0.001645) | Val Total: 0.002091 (f: 0.000251, f': 0.001840)
Epoch [600/1000] Train Total: 0.000004 (f: 0.000000, f': 0.000004) | Val Total: 0.000005 (f: 0.000000, f': 0.000005)
Epoch [800/1000] Train Total: 0.000006 (f: 0.000000, f': 0.000006) | Val Total: 0.000008 (f: 0.000000, f': 0.000007)
Epoch [1000/1000] Train Total: 0.000002 (f: 0.000000, f': 0.000002) | Val Total: 0.000003 (f: 0.000000, f': 0.000003)
```



- Training and validation losses both decrease smoothly
- The network's prediction closely match the true

Runge function .

```

import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split
torch.manual_seed(0)
np.random.seed(0)

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

def runge_derivative_np(x_np):
    # f'(x) = -50x / (1+25 x^2)^2 (numpy)
    return -(50.0 * x_np) / (1.0 + 25.0 * x_np**2)**2

#dataset
x = np.linspace(-1, 1, 200).reshape(-1, 1).astype(np.float32)
y = runge_function(x).astype(np.float32)
x_train, x_val, y_train, y_val = train_test_split(
    x, y, test_size=0.2, random_state=42
)

x_train_t = torch.from_numpy(x_train)
y_train_t = torch.from_numpy(y_train)
x_val_t = torch.from_numpy(x_val)
y_val_t = torch.from_numpy(y_val)

dy_train_t = torch.from_numpy(runge_derivative_np(x_train).astype(np.float32))
dy_val_t = torch.from_numpy(runge_derivative_np(x_val).astype(np.float32))

#model
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(1, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 1)
        self.activation = nn.Tanh()
    def forward(self, x):
        x = self.activation(self.fc1(x))
        x = self.activation(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net()

#Loss:= function loss + derivative loss
mse = nn.MSELoss()

train_loss_all, val_loss_all = [], []
train_loss_f, train_loss_df = [], []
val_loss_f, val_loss_df = [], []

lambda_deriv = 1.0
lr = 0.01
optimizer = optim.Adam(net.parameters(), lr=lr)
epochs = 1000

for epoch in range(epochs):
    net.train()
    optimizer.zero_grad()

    x_train_req = x_train_t.detach().clone().requires_grad_(True)

    y_hat = net(x_train_req)

    dy_hat = torch.autograd.grad(
        outputs = y_hat,
        inputs = x_train_req,
        grad_outputs = torch.ones_like(y_hat),
        create_graph = True,
        retain_graph = True
    )[0]

    loss_f = (y_hat - y_train_t).pow(2).mean()
    loss_df = (dy_hat - dy_train_t).pow(2).mean()
    loss = loss_f + lambda_deriv * loss_df

    loss.backward()
    optimizer.step()

    net.eval()
    with torch.no_grad():
        y_val_hat = net(x_val_t)
        loss_f_val = (y_val_hat - y_val_t).pow(2).mean().item()

    x_val_req = x_val_t.detach().clone().requires_grad_(True)
    y_val_hat2 = net(x_val_req)
    dy_val_hat = torch.autograd.grad(
        outputs = y_val_hat2,
        inputs = x_val_req,
        grad_outputs = torch.ones_like(y_val_hat2),
        create_graph = False,
        retain_graph = False
    )[0].detach()
    loss_df_val = (dy_val_hat - dy_val_t).pow(2).mean().item()

    train_loss_all.append(loss.item())
    val_loss_all.append(loss_f_val + lambda_deriv * loss_df_val)
    train_loss_f.append(loss_f.item())
    train_loss_df.append(loss_df.item())
    val_loss_f.append(loss_f_val)
    val_loss_df.append(loss_df_val)

    if (epoch + 1) % 200 == 0:
        print(f"Epoch {epoch+1}/{epochs} | "
              f"Train Total: {train_loss_all[-1]:.6f} | "
              f"({train_loss_f[-1]:.6f}, {train_loss_df[-1]:.6f}) | "
              f"Val Total: {val_loss_all[-1]:.6f} | "
              f"({val_loss_f[-1]:.6f}, {val_loss_df[-1]:.6f})")

#true f(x) vs NN prediction
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(x_test, y_test, label="True f(x)")
plt.plot(x_test, net(torch.from_numpy(x_test)).detach().numpy(),
         "--", label="NN prediction")
plt.xlabel("x"); plt.ylabel("f")
plt.title("True vs NN Prediction")
plt.legend()
plt.tight_layout()
plt.show()

#training/validation total loss curves
plt.figure(figsize=(12,5))
plt.subplot(1,2,2)
plt.plot(train_loss_all, label="Train Total Loss")
plt.plot(val_loss_all, label="Val Total Loss")
plt.xlabel("Epoch"); plt.ylabel("Loss")
plt.title("Training/Validation Loss Curves")
plt.legend()
plt.tight_layout()
plt.show()

#final test MSE
criterion = nn.MSELoss()
with torch.no_grad():
    test_mse_f = criterion(torch.from_numpy(y_test),
                           net(torch.from_numpy(x_test))).item()
    x_test_req2 = torch.from_numpy(x_test).requires_grad_(True)
    y_hat_t = net(x_test_req2)
    dy_hat_t = torch.autograd.grad(
        outputs = y_hat_t,
        inputs = x_test_req2,
        grad_outputs = torch.ones_like(y_hat_t),
        create_graph = False,
        retain_graph = False
    )[0].detach()
    test_mse_df = criterion(dy_hat_t, torch.from_numpy(dy_true)).item()

print(f"Final Test MSE (f): {test_mse_f:.6f}")
print(f"Final Test MSE (f'): {test_mse_df:.6f}")

```