114-1 Machine Learning

Week 3 Assignment

Ming Hsun Wu

September 26, 2025

Problem 1.

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

Lemma 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*

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

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

Lemma 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*

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

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

NOTE OF PROBLEM 1.

1 Lemma 1 :

Lemma statement:

Lemma 1 tells us that for any odd-degree polynomial, it is possible to find a shallow tanh neural network that approximates it. Moreover, not only can the function itself be approximated closely, but derivatives of any order can also be approximated.

Why we can do it?:

The key lies in the tanh tanh function. From calculus, we know through the Taylor expansion that many functions can be expressed as a series in powers of x. The function $tanh(x)$ is special because its expansion contains only odd-degree terms:

$$tanh(x) = x - \frac{x^3}{3} + \frac{2x^5}{15} - \frac{17x^7}{315} + ...$$

This makes $tanh(x)$ particularly suitable for approximating odd-power functions. By choosing appropriate scalings and combining several $tanh$ functions, one can effectively approximate various odd-degree terms.

2 Lemma 2 :

Lemma statement:

Lemma 2 further tells us that for any polynomial, there exists a neural network composed of *tanh* functions that can approximate it. The odd-degree terms are approximated in the same manner as described in Lemma 1, while the even-degree terms can also be approximated by leveraging the method from Lemma 1. Similarly, not only can the function itself be approximated with sufficient accuracy, but derivatives of any order can also be approximated.

Why we can do it?:

As stated in Lemma 1, we can approximate odd-degree polynomials. Lemma 2 further extends this result, showing that for polynomials of any degree, there exists a neural network composed of *tanh*functions that can approximate them. The key idea is to approximate even-degree terms using odd-degree terms through finite difference methods (see the formula below). Once the odd-degree terms are obtained, Lemma 1 can then be applied to establish Lemma 2.

$$(\psi)_{2n}(y) = \frac{1}{2\alpha(2n+1)}\Big(\hat{f}_{2n+1,h}(y+\alpha) - \hat{f}_{2n+1,h}(y-\alpha) - 2\sum_{k=0}^{n-1}\binom{2n+1}{2k}\alpha^{2(n-k)+1}(\psi)(y)_{2k}\Big)$$

This explains why the network in Lemma 2 needs to be larger, since additional units are required to handle the even-degree terms.

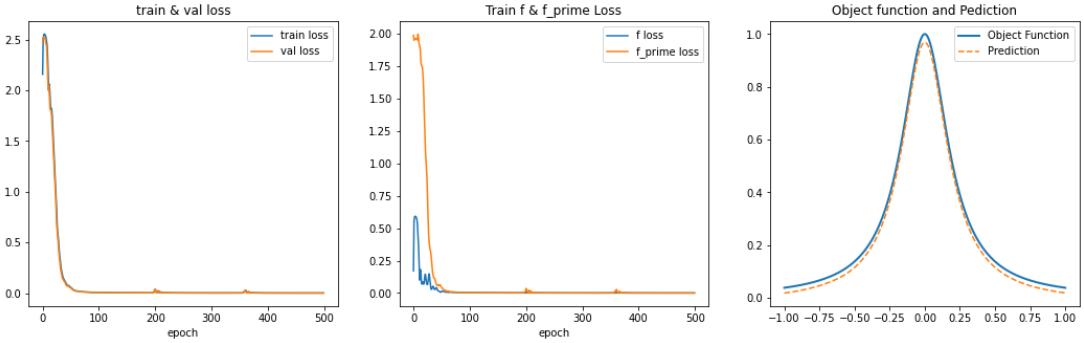
Problem 2.

If possible, I hope the teacher can write a little slower, because sometimes it is hard to tell what the teacher is writing...

PROBLEM 3.

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

SOLUTION.



NOTE OF PROBLEM 3.

About my hypothesis	
training data	300 equally spaced points in the interval $[1, 1]$.
testing data	500 equally spaced points in the interval $[1, 1]$.
$f_\theta : \mathbb{R} \to \mathbb{R}$	$f_\theta = t_3(L_3(t_2(L_2(t_1(L_1(x))))))$
	where $t_i = tanh(x)$ $L_i = W_i x + b_i$
Hidden layer and activation function	3 *linear layer* with *tanch*
Trainable params	8,577
Loss function	$mse(f_\theta(x), y) + mse(f'_\theta(x), y')$, y is true data
epoch	500

Following the HW2 code, I modified the training and validation processes by adding a step that computes the derivatives of my model's outputs using torch.autograd.grad. I then combined these results with the Runge function to calculate both the function loss and the derivative loss. The sum of these two losses was used as my defined loss function to update the model parameters.

For the trained model, the MSE on the test data is 0.000840.

```
#%% source code (The only difference between HW3 and HW2 is the
    definition of LOSS)
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import torch
import torch.optim as optim
from torchinfo import summary
#%%
'data preprocessing'

# obkect function
def object_function(x):
    return 1 / (1 + 25 * x**2)

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


x = np.linspace(-1, 1, 5000).reshape(-1, 1).astype(np.float32)
y = object_function(x).astype(np.float32)
y_prime = runge_derivative(x).astype(np.float32)
# train:0.7 val:0.2
x_train, x_val, y_train, y_val, yprime_train, yprime_val =
    train_test_split(
```

```
    x, y, y_prime, test_size=0.2, random_state=42
)
# data transform to tensor
x_train = torch.tensor(x_train, requires_grad=True)
y_train = torch.tensor(y_train)
yprime_train = torch.tensor(yprime_train)

x_val = torch.tensor(x_val, requires_grad=True)
y_val = torch.tensor(y_val)
yprime_val = torch.tensor(yprime_val)

#%%
'create network'
class Curve_Fitting(torch.nn.Module):
    def __init__(self):
        super(Curve_Fitting, self).__init__()
        self.Linear_1 = torch.nn.Linear(1, 128)
        self.Linear_2 = torch.nn.Linear(128, 64)
        self.Linear_3 = torch.nn.Linear(64, 1)
        self.Tanh = torch.nn.Tanh()

    def forward(self, x):
        x = self.Linear_1(x)
        x = self.Tanh(x)
        x = self.Linear_2(x)
        x = self.Tanh(x)
        x = self.Linear_3(x)
```

```
52          x = self.Tanh(x)
53
54          return x
55
56  # compile model and loss
57  model = Curve_Fitting()
58  MSE = torch.nn.MSELoss()
59  optimizer = optim.Adam(model.parameters(), lr=0.01)
60
61  def custom_loss(x, y_true, yprime_true):
62      y_pred = model(x)
63      func_loss = torch.nn.MSELoss()(y_pred, y_true)
64
65      dydx = torch.autograd.grad(
66          outputs=y_pred,
67          inputs=x,
68          grad_outputs=torch.ones_like(y_pred),
69          create_graph=True
70      )[0]
71
72      deriv_loss = torch.nn.MSELoss()(dydx, yprime_true)
73      return func_loss + deriv_loss, func_loss.item(),
            deriv_loss.item()
74  #%%
75  'train model and predict'
76  epochs = 500
77  train_losses, val_losses = [], []
```

```
78 func_losses, deriv_losses = [], []
79 for epoch in range(epochs):
80     # train
81     model.train()
82     optimizer.zero_grad()
83     loss, f_loss, d_loss = custom_loss(x_train, y_train,
           yprime_train)
84     loss.backward()
85     optimizer.step()
86     train_losses.append(loss.item())
87     func_losses.append(f_loss)
88     deriv_losses.append(d_loss)
89
90     # val
91     model.eval()
92
93     y_pred_val = model(x_val)
94     dydx_val = torch.autograd.grad(
95         outputs=y_pred_val,
96         inputs=x_val,
97         grad_outputs=torch.ones_like(y_pred_val),
98         create_graph=True
99     )[0]
100    val_loss = torch.nn.MSELoss()(y_pred_val, y_val) +
           torch.nn.MSELoss()(dydx_val, yprime_val)
101    val_losses.append(val_loss.item())
102
```

```
# estimation
x_test = torch.tensor(np.linspace(-1, 1, 500).reshape(-1,
    1).astype(np.float32), requires_grad=True)
with torch.no_grad():
    y_pred = model(x_test).numpy()
y_true = object_function(x_test.detach().numpy())
mse = mean_squared_error(y_true, y_pred)
print(f"The MSE of predict and true data: {mse:.6f}")


#%%
'plot result'
plt.figure(figsize=(18,5))
# loss
plt.subplot(1,3,1)
plt.plot(train_losses, label="train loss")
plt.plot(val_losses, label="val loss")
plt.xlabel('epoch')
plt.legend()
plt.title("train & val loss")

plt.subplot(1,3,2)
plt.plot(func_losses, label="f loss")
plt.plot(deriv_losses, label="f_prime loss")
plt.xlabel('epoch')
plt.legend()
plt.title("Train f & f_prime Loss")
```

```
# function
plt.subplot(1,3,3)
plt.plot(x_test.detach().numpy(), y_true, label="Object Function",
    linewidth=2)
plt.plot(x_test.detach().numpy(), y_pred, label="Prediction",
    linestyle="--")
plt.legend()
plt.title("Object function and Pediction")

plt.show()
```