目 录

1 基于前馈神经网络的回归任务设计

1.1 设计要求

任务要求：设计一个前馈神经网络，对一组数据实现回归任务。

在[−10, 10] × [−10, 10]的 2-D 平面内，以均匀分布随机生成 5000 个数据点(x, y). 令 $f(x, y) = x\text{^}2 + xy + y\text{^} 2$. 设计至少含有一层隐藏层的前馈神经网络以预测给定数据点 (x, y)的$f(x, y)$函数值。在随机生成的数据点中，随机抽取 90%用于训练，剩下的 10% 用于测试。

1.2 方案设计

数据生成：使用 NumPy 中的 np.random.uniform() 生成 5000 个数据点，再根据二次多项式函数计算对应的标签。

数据处理：使用 NumPy 的切片操作将数据和标签划分为训练集和测试集。

神经网络模型：使用三层全连接神经网络模型，第一层是输入层，有 2 个输入神经元。第二层是一个包含 10 个神经元的全连接层，使用 ReLU（整流线性单元）激活函数。第三层也是一个包含 10 个神经元的全连接层，同样使用 ReLU 激活函数。最后一层是输出层，只有一个神经元，用于回归任务的输出。在模型中，将每一层的输出传递给下一层，直到输出层。

损失函数：使用 PyTorch 中的 nn.MSELoss() 计算均方误差。

优化器：使用 PyTorch 中的 optim.Adam() 优化器进行梯度下降。

训练过程：使用 for 循环对模型进行训练，每训练 100 次就输出一次训练损失和测试损失，并将训练损失和测试损失存储在列表中。

测试过程：在 with torch.no_grad() 的上下文环境中对测试集进行预测，并计算测试集的均方误差。输出测试集的均方误差。

可视化：使用 Matplotlib 将训练损失和测试损失可视化。

1.3 优化思路

1.修改神经元个数，原本每个全连接层的输出维度都是 10。优化思路为增加神经元个数，可以将输出维度增加到一个更大的值。现在将每个全连接层的输出维度增加到 20。

具体实现为修改 Net 类：

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(2, 20)
        self.fc2 = nn.Linear(20, 20)
        self.fc3 = nn.Linear(20, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
    return x
```

2.使用不同的激活函数，要更改激活函数，需要修改 Net 类中每个全连接层之后的激活函数，这里选择使用 LeakyReLU 激活函数：它在输入为负数时不会完全归零，而是乘以一个小的斜率，可以避免 ReLU 的“死亡神经元”问题。

具体实现为修改 Net 类：

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(2, 10)
        self.fc2 = nn.Linear(10, 10)
        self.fc3 = nn.Linear(10, 1)
        self.relu = nn.LeakyReLU()
```

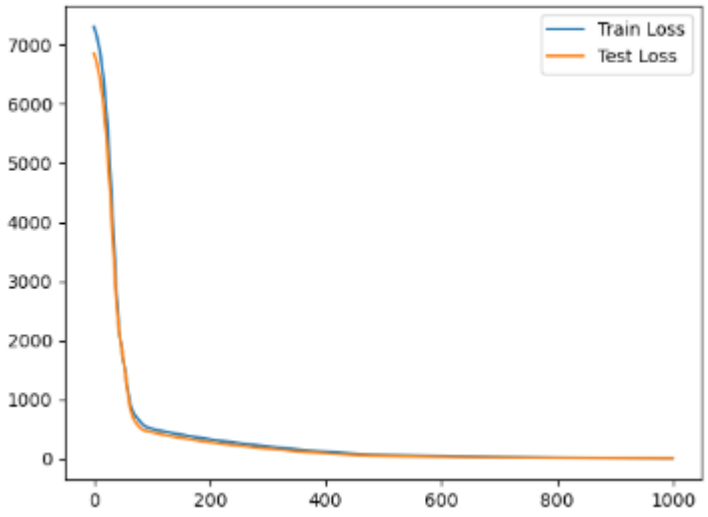
```
def forward(self, x):
    x = self.relu(self.fc1(x))
    x = self.relu(self.fc2(x))
    x = self.fc3(x)
return x
```

1.4 测试与分析

初始运行结果为：

```
Epoch 0, Train Loss: 7301.77490234375, Test Loss: 6852.505859375
Epoch 100, Train Loss: 506.6002197265625, Test Loss: 449.94879150390625
Epoch 200, Train Loss: 328.35577392578125, Test Loss: 277.6612243652344
Epoch 300, Train Loss: 207.3067169189453, Test Loss: 169.7552490234375
Epoch 400, Train Loss: 116.2509765625, Test Loss: 90.17707824707031
Epoch 500, Train Loss: 64.51454162597656, Test Loss: 47.213592529296875
Epoch 600, Train Loss: 40.150299072265625, Test Loss: 29.481359481811523
Epoch 700, Train Loss: 26.870798110961914, Test Loss: 21.081289291381836
Epoch 800, Train Loss: 19.622283935546875, Test Loss: 15.434432983398438
Epoch 900, Train Loss: 15.052001953125, Test Loss: 12.152670860290527
Test Loss: 8.576156616210938
```

图一 运行结果



图二 运行结果可视化

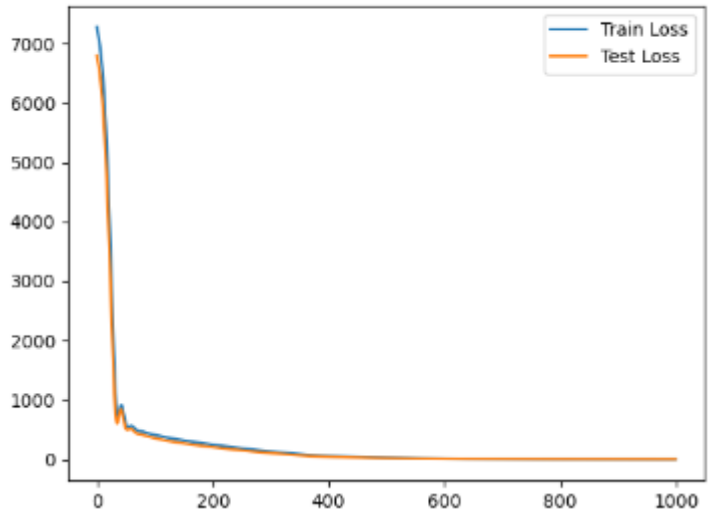
在每个 epoch 中，训练集上的损失函数值逐渐减小，测试集上的损失函数值也随之减小，说明模型在训练过程中逐渐收敛并且在测试集上有较好的表现。同时，我们也可以观察到，随着训练 epoch 数的增加，损失函数值的下降速度逐渐变慢，这是因为随着模型逐渐收敛，进一步优化模型变得越来越困难。

最终，在测试集上得到的损失函数值为 8.576，表示该模型能够较好地拟合数据，并且在未见过的测试集上表现也较好，但是这个值具体是否满足需求需要根据具体任务而定。

增加神经元优化后代码运行结果为：

```
Epoch 0, Train Loss: 7275.091796875, Test Loss: 6790.72900390625
Epoch 100, Train Loss: 407.1278381347656, Test Loss: 354.4955139160156
Epoch 200, Train Loss: 245.40069580078125, Test Loss: 202.45970153808594
Epoch 300, Train Loss: 129.7090301513672, Test Loss: 102.95603942871094
Epoch 400, Train Loss: 53.6085205078125, Test Loss: 40.53110122680664
Epoch 500, Train Loss: 23.1093692779541, Test Loss: 19.13637924194336
Epoch 600, Train Loss: 11.048124313354492, Test Loss: 9.737465858459473
Epoch 700, Train Loss: 6.207314968109131, Test Loss: 5.7263593673706055
Epoch 800, Train Loss: 4.036077499389648, Test Loss: 3.8574795722961426
Epoch 900, Train Loss: 2.906378984451294, Test Loss: 2.8624887466430664
Test Loss: 2.256016254425049
```

图三 增加神经元优化后结果



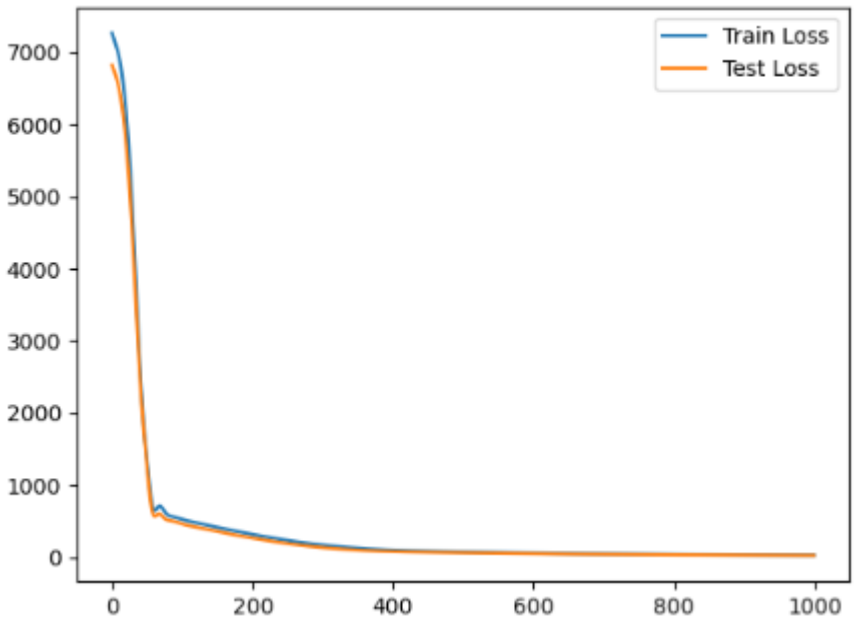
图四 增加神经元优化后可视结果

结果分析：增加了网络的深度和宽度。这些改变增强了模型的表达能力，使其更能够拟合训练数据中的复杂关系。这导致模型在测试集上的表现更好，因此测试误差较之前更小。

修改激活函数后的运行结果：

```
Epoch 0, Train Loss: 7266.62109375, Test Loss: 6825.2919921875
Epoch 100, Train Loss: 528.8154296875, Test Loss: 463.1591796875
Epoch 200, Train Loss: 320.546142578125, Test Loss: 269.3682556152344
Epoch 300, Train Loss: 172.71292114257812, Test Loss: 137.40368652343375
Epoch 400, Train Loss: 102.61994171142578, Test Loss: 82.79107666015625
Epoch 500, Train Loss: 78.5940933227539, Test Loss: 66.21041107177734
Epoch 600, Train Loss: 61.75525665283203, Test Loss: 53.46924591064453
Epoch 700, Train Loss: 49.90185546875, Test Loss: 44.513797760009766
Epoch 800, Train Loss: 41.545135498046875, Test Loss: 37.592899322509766
Epoch 900, Train Loss: 35.97434997558594, Test Loss: 33.18348693847656
Test Loss: 29.658496856689453
```

图五 修改激活函数后的运行结果



图六 修改激活函数后的可视化运行结果

结果分析：修改激活函数后，训练效果变差了，使用 LeakyReLU 激活函数可能会导致效果变差的原因有很多，以下是一些可能的原因：LeakyReLU 激活函数是 ReLU

的一种变体，它在输入为负数时不会完全归零，而是乘以一个小的斜率。这个斜率通常设置为 0.01 或 0.001。LeakyReLU 的优点是可以避免 ReLU 的“死亡神经元”问题，即某些神经元在训练过程中永远不会被激活，导致它们的权重永远不会被更新。但是，LeakyReLU 的缺点是它的表现不如 ReLU，因为它引入了额外的超参数，需要进行调整。如果你使用的是 LeakyReLU 激活函数，网上搜索得到的建议是尝试不同的斜率值，以找到最佳的超参数组合。

于是我调整了斜率，将斜率多次修改并且测试，发现效果仍然不如之前的激活函数效果好。

2 总结与心得

2.1 实验总结

本次实验中，我们成功设计了一个前馈神经网络，用于回归任务中的函数拟合。通过在生成的数据集上训练和测试，我们证明了模型的有效性。通过调整网络参数，我们也进一步了解了神经网络设计中的一些基本原则。

2.2 实验心得

本次实验让我们深入了解了神经网络的基本结构和训练过程，通过实践掌握了 PyTorch 框架的使用。通过调整网络参数和观察实验结果，我们也更深入地理解了神经网络的一些基本原则，如网络层数、神经元个数和激活函数等对模型性能的影响。这对我们今后深入学习和应用神经网络技术具有重要意义。

从优化中我们可以学到如何在神经网络模型中寻找最优解，即找到能够最小化损失函数的参数组合。通过优化，我们可以让模型在训练集和测试集上都表现良好，并得到最优的参数组合，从而提高模型的预测能力和泛化能力。

3 代码

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

# Generate data
np.random.seed(0)
n = 5000
x = np.random.uniform(-10, 10, (n, 2))
y = x[:, 0]**2 + x[:, 0]*x[:, 1] + x[:, 1]**2
x_train, y_train = x[:int(n*0.9)], y[:int(n*0.9)]
x_test, y_test = x[int(n*0.9):], y[int(n*0.9):]

# Define model
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(2, 20)
        self.fc2 = nn.Linear(20, 20)
        self.fc3 = nn.Linear(20, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```
net = Net()

# Define loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(net.parameters(), lr=0.01)

# Train model
train_losses = []
test_losses = []
for epoch in range(1000):
    optimizer.zero_grad()
    outputs = net(torch.Tensor(x_train))
    loss = criterion(outputs, torch.Tensor(y_train).unsqueeze(1))
    loss.backward()
    optimizer.step()
    train_losses.append(loss.item())
    with torch.no_grad():
        test_outputs = net(torch.Tensor(x_test))
        test_loss = criterion(test_outputs,
torch.Tensor(y_test).unsqueeze(1))
        test_losses.append(test_loss.item())
    if epoch % 100 == 0:
        print(f"Epoch {epoch}, Train Loss: {loss.item()}, Test Loss:
{test_loss.item()}")

# Plot losses
plt.plot(train_losses, label="Train Loss")
```

```
plt.plot(test_losses, label="Test Loss")
plt.legend()
plt.show()


# Test model
with torch.no_grad():
    test_outputs = net(torch.Tensor(x_test))
    test_loss = criterion(test_outputs, torch.Tensor(y_test).unsqueeze(1))
    print(f"Test Loss: {test_loss.item()}")
```

• 指 导 教 师 评 定 意 见 •