# 深度学习 实验1 前馈神经网络

PB21081601 张芷苒

## 实验要求

使用 pytorch 或者 tensorflow 手写一个前馈神经网络, 用于近似以下函数:

$$y = \log_2(x) + \cos\left(\frac{\pi x}{2}\right), x \in [1, 16]$$

研究**数据量、网络深度、学习率、网络宽度、激活函数**对模型性能的影响。

## 实验步骤

### 网络框架

本次实验我选择了pytorch作为框架。

### 数据生成

按照要求，数据集使用程序自动生成, 即在取值范围内随机采样作为 $x$ 值, 并计算相应的 $y$ 值。要求在 $x \in [1, 16]$ 范围内均匀采样获得 $N$ 个样本的数据集, 再按训练集: 验证集: 测试集=8: 1: 1 的比例随机划分成三个互不相交的部分，并且在调参分析中固定三个部分的组成。

```
# 生成和预处理数据
x = np.linspace(1, 16, num=10000)
y = target_function(x)
x_tensor = torch.tensor(x.reshape(-1, 1), dtype=torch.float32) # 将x转换为张量
y_tensor = torch.tensor(y.reshape(-1, 1), dtype=torch.float32) # 将y转换为张量
X_train, X_temp, y_train, y_temp = train_test_split(x_tensor, y_tensor, test_size=0.2, random_state=42) # 划分训练集和剩余数据
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42) # 划分验证集和测试集
```

### 模型搭建

1. **继承与初始化**：
    - `class Net(nn.Module)`：定义了一个名为 `Net` 的新类，它继承自 `torch.nn.Module`。`torch.nn.Module` 是构建所有神经网络的基类。

- `super(Net, self).__init__()`：调用父类 `nn.Module` 的构造函数来执行必要的初始化。

2. **层的定义**：
   - `self.fc1 = nn.Linear(1, 256)`：定义了第一个全连接层（也称为线性层），它将接受单个输入特征，并输出256个特征。这作为网络的输入层。
   - `self.fc2` 和 `self.fc3`：这两行定义了更多的全连接层，每层都有256个输入和输出特征，作为隐藏层来进一步处理数据。
   - `self.fc4 = nn.Linear(256, 1)`：定义了最后一个全连接层，它将之前层的输出（256个特征）转换为单个输出值。这是网络的输出层，用于回归预测。
3. **前向传播**：
   - `def forward(self, x)`：定义了数据通过网络的前向传播路径。在PyTorch中，你需要重写 `nn.Module` 的 `forward` 方法来定义模型的输出行为。
   - 在每个全连接层后，使用了ReLU激活函数（`F.relu`），它添加了非线性，使得网络能够学习和模拟更复杂的输入到输出的映射关系。
   - 最后一个全连接层后没有使用激活函数，直接返回了其输出，因为我们希望模型能够预测任意范围内的连续值。
4. **模型实例化**：
   - `model = Net()`：使用定义的 `Net` 类创建了一个模型实例。这个实例化的模型 `model` 现在可以用于训练和预测。

```
# 定义模型
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(1, 256)  # 输入层到第一个隐藏层
        self.fc2 = nn.Linear(256, 256) # 第一个隐藏层到第二个隐藏层
        self.fc3 = nn.Linear(256, 256) # 新增加 第二个隐藏层到第三个隐藏层
        self.fc4 = nn.Linear(256, 1)  # 第三个隐藏层到输出层

    def forward(self, x):
        x = F.relu(self.fc1(x))  # 第一个隐藏层激活
        x = F.relu(self.fc2(x))  # 第二个隐藏层激活
        x = F.relu(self.fc3(x))  # 第三个隐藏层激活
        return self.fc4(x)  # 输出层，没有激活函数

model = Net()
```

## 模型训练

1. **损失函数**:

- `criterion = nn.MSELoss()`：定义损失函数为均方误差（Mean Squared Error, MSE）。

2. **优化器**:
   - `optimizer = torch.optim.Adam(model.parameters(), lr=0.009)`：选择Adam优化器来更新模型的权重，`model.parameters()`获取模型中所有可训练的参数（权重和偏置），`lr=0.009`设置学习率为0.009。学习率控制参数更新步长。
3. **训练循环**:
   - 循环进行15000次迭代（epochs），每轮迭代中：
     - `optimizer.zero_grad()`：清零之前的梯度。
     - `output = model(X_train)`：计算模型对训练数据`X_train`的输出预测。
     - `loss = criterion(output, y_train)`：计算预测值`output`和实际值`y_train`之间的损失。
     - `loss.backward()`：执行反向传播，根据损失计算每个参数的梯度。
     - `optimizer.step()`：根据计算得到的梯度更新模型的参数。
4. **监控训练进度**:
   - 每1000个epochs输出一次当前的损失值。

```
# 训练模型
criterion = nn.MSELoss() # 定义损失函数
optimizer = torch.optim.Adam(model.parameters(), lr=0.009) # 定义优化器


for epoch in range(15000):
    optimizer.zero_grad() # 梯度清零
    output = model(X_train)
    loss = criterion(output, y_train)
    loss.backward() # 反向传播
    optimizer.step()

    if epoch % 1000 == 0:
        print(f'Epoch {epoch+1}, Loss: {loss.item()}')
```

## 调参分析

在最初的代码中，我进行了最简单的前馈神经网络架构，超参分别取值：
网络深度 3层前馈神经网络
学习率 0.01
网络宽度 64
激活函数 ReLU

为了获得更低的MSE，在数据量10000的情况下，对这四个超参进行调整。

## 网络深度

将网络深度从3层加到4层，loss有所下降。但是当加到5层时，随着迭代周期变多，loss值下降不稳定，下降也无显著差别。因此确定网络深度为4层。

## 学习率

初始学习率设置为了0.01，在降低为0.009之后损失函数波动没有那么剧烈，因此可以认为稍微降低学习率有利于损失函数平稳下滑。但是降低为0.008后没有进一步改善。故确定学习率为0.009.

## 网络宽度

64的网络宽度可能导致学习能力有限，增加至128和256时损失率均有下降，但是再增加会导致过拟合风险，因此确定网络宽度为128。

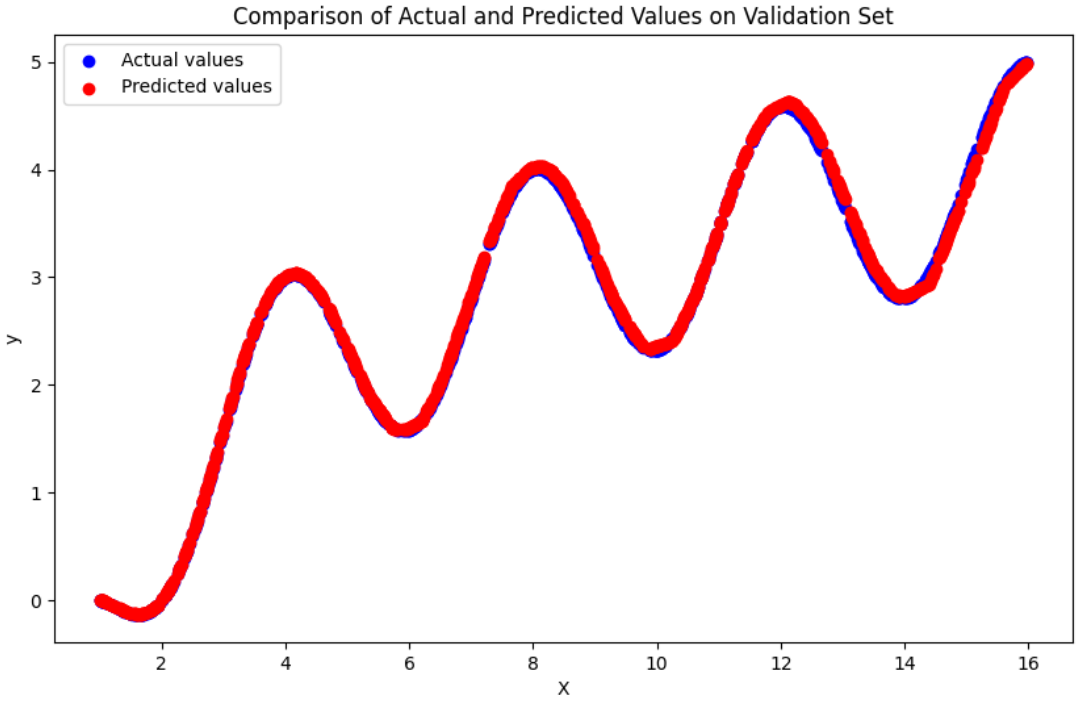
## 激活函数

将激活函数更换后并没有发现损失率有明显下降，因此认为ReLU这个激活函数的缺陷没有在对损失率的影响中有过多效果。不更换激活函数。

# 验证集验证

在这组超参下进行验证集上的测试，得到如下结果：

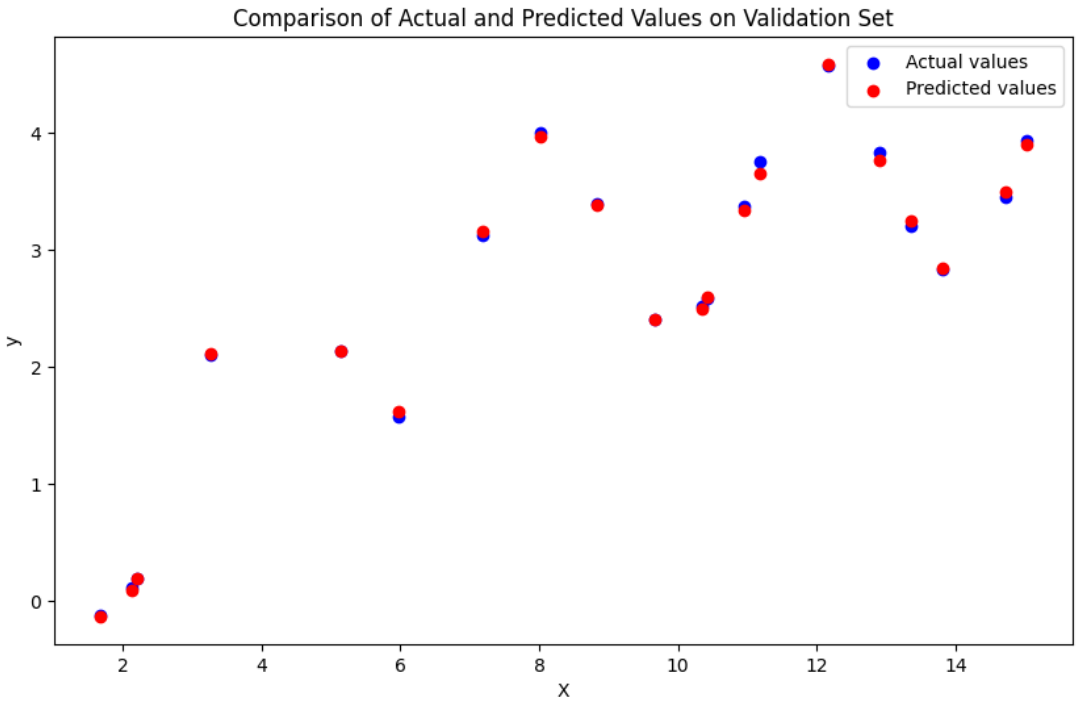
```
Epoch 1, Loss: 10.755262374877930
Epoch 1001, Loss: 0.34507361054420470
Epoch 2001, Loss: 0.10477117449045181
Epoch 3001, Loss: 0.00344814243726432320
Epoch 4001, Loss: 0.00515191163867712
Epoch 5001, Loss: 0.013787410221993923
Epoch 6001, Loss: 0.010187542997300625
Epoch 7001, Loss: 0.000835440878290683
Epoch 8001, Loss: 0.00042061760905198753
Epoch 9001, Loss: 0.00029271299717947840
Epoch 10001, Loss: 0.003287993837147951
Epoch 11001, Loss: 0.00030727643752470610
Epoch 12001, Loss: 0.0026729817036539316
Epoch 13001, Loss: 0.00025398732395842670
Epoch 14001, Loss: 0.00020619670976884663
```




这个结果相比于初始参数的损失率，确实有显著下降。

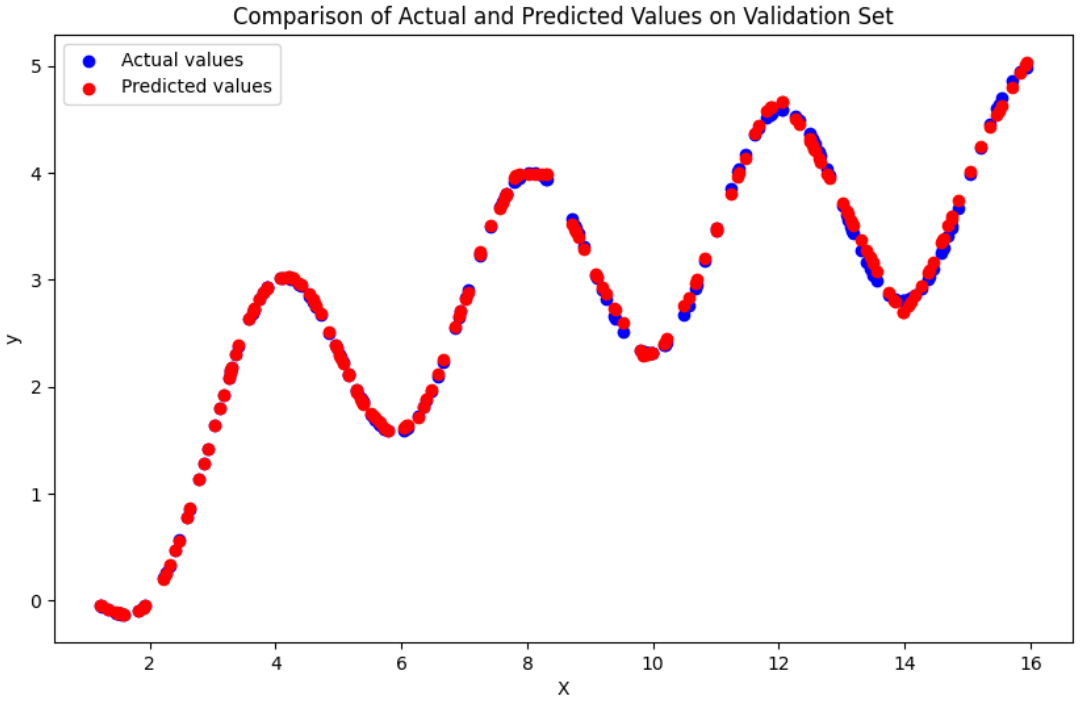
对数据量200进行验证得到结果：

```
poch 1, Loss: 10.0354318618774 41
Epoch 1001, Loss: 0.2133231908082962
Epoch 2001, Loss: 0.22440800070762634
Epoch 3001, Loss: 0.20806849002838135
Epoch 4001, Loss: 0.29684701561927795
Epoch 5001, Loss: 0.25758326053619385
Epoch 6001, Loss: 0.18877694010734558
Epoch 7001, Loss: 0.10566385090351105
Epoch 8001, Loss: 0.0136437295004725 46
Epoch 9001, Loss: 0.003188739065080881
Epoch 10001, Loss: 0.0046439631842076 78
Epoch 11001, Loss: 0.0026689814403653145
Epoch 12001, Loss: 0.005907848011702299
Epoch 13001, Loss: 0.002298962092027068
Epoch 14001, Loss: 0.002524628769606352
```




对数据量2000进行验证得到结果：

```
Epoch 1, Loss: 12.742654800415039
Epoch 1001, Loss: 0.3699113130569458
Epoch 2001, Loss: 0.3233443796634674
Epoch 3001, Loss: 0.26941290497779846
Epoch 4001, Loss: 0.22407230734825134
Epoch 5001, Loss: 0.2226649522781372
Epoch 6001, Loss: 0.18658533692359924
Epoch 7001, Loss: 0.18645647168159485
Epoch 8001, Loss: 0.10673842579126358
Epoch 9001, Loss: 0.008565220981836319
Epoch 10001, Loss: 0.009086334146559238
Epoch 11001, Loss: 0.007584318518638611
Epoch 12001, Loss: 0.042972203344106674
Epoch 13001, Loss: 0.002165034646168351
Epoch 14001, Loss: 0.020138952881097794
```




注意，对于其他两个数据量，这组超参不是使其在验证集上表现最好的超参。由于时间有限，原理相同，在此不再进行调参分析。

## 测试性能

经过调参分析，在数据量为10000时，确定超参为：
网络深度 4层前馈神经网络
学习率 0.009
网络宽度 256
激活函数 ReLU
时，在验证集上的表现较好，MSE loss可以达到0.0003-0.0006之间。

重新训练模型，在测试集上测试。测试集代码如下：

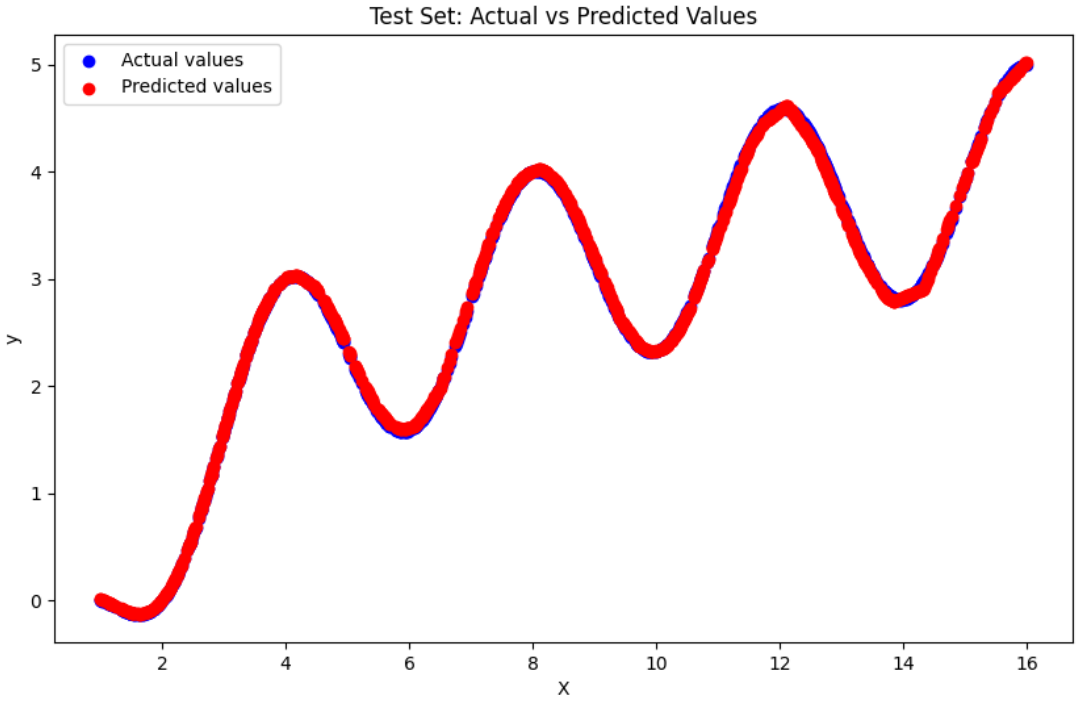
```
# 模型评估 - 使用测试集
model.eval()  # 将模型设置为评估模式
with torch.no_grad():  # 关闭梯度计算
    predictions_test = model(X_test)  # 在测试集上进行预测
    test_loss = criterion(predictions_test, y_test)  # 计算测试集上的损失
    print(f'Test Loss: {test_loss.item():.4f}')  # 打印测试集损失

# 可视化测试集的实际值和预测值
plt.figure(figsize=(10, 6))
plt.scatter(X_test.numpy(), y_test.numpy(), color='blue', label='Actual
values')
plt.scatter(X_test.numpy(), predictions_test.numpy(), color='red',
label='Predicted values')
```

```
plt.title('Test Set: Actual vs Predicted Values')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()
```

得到的输出结果：

```
Epoch 1, Loss: 8.720077514648438
Epoch 1001, Loss: 0.21444958448410034
Epoch 2001, Loss: 0.40935373306274414
Epoch 3001, Loss: 0.035440370440048309
Epoch 4001, Loss: 0.016503822058439255
Epoch 5001, Loss: 0.0036771618761122227
Epoch 6001, Loss: 0.0012992491247132 42
Epoch 7001, Loss: 0.025860384106636047
Epoch 8001, Loss: 0.000807550793979317
Epoch 9001, Loss: 0.00284168915823102
Epoch 10001, Loss: 0.004834367427974939
Epoch 11001, Loss: 0.0006330112228170037
Epoch 12001, Loss: 0.005084204953163862
Epoch 13001, Loss: 0.00020987399329897016
Epoch 14001, Loss: 0.000307468551909551
Test Loss: 0.0005
```



对于其他数据量的测试，这一组超参依然可以满足test loss在0.002以下。但是对于这两个数据量，这组超参可能不是使其在验证集上表现最好的超参。由于时间有限，原理相同，在此不再赘述。

# 实验收获

本次实验使我熟悉了前馈神经网络的构建与训练过程。
在数据量固定时，对超参进行分析和控制变量的调整以提升拟合度的过程使我加深了对前馈神经网络原理的理解。