使用两层 ReLU 神经网络拟合自定义函数

1. 问题描述

理论和实验证明，一个两层的 ReLU 网络可以模拟任何函数[1~5]。本报告的目标是自行定义一个函数，并使用基于 ReLU 的神经网络来拟合此函数。我们将通过生成训练集和测试集来训练神经网络，并使用测试集来验证拟合效果。

2. 函数定义

我们定义的目标函数如下：

```
def target_function(x):
    return np.sin(np.pi * x) + 0.5 * np.cos(2 * np.pi * x)
```

3. 数据采集

我们使用 np.random.uniform 在区间 [−1,1] 上随机生成 1000 个训练样本，并使用 np.linspace 在相同区间上生成 100 个均匀分布的测试样本。

```
np.random.seed(42)
x_train = np.random.uniform(-1, 1, 10000).reshape(-1, 1)
y_train = target_function(x_train)
x_test = np.linspace(-1, 1, 1000).reshape(-1, 1)
y_test = target_function(x_test)
```

4. 模型描述

我们使用了一个两层的 ReLU 神经网络，结构如下：

① 输入层：1 个神经元（输入特征为 1 维）

② 隐藏层：256 个神经元，激活函数为 ReLU

③ 输出层：1 个神经元（输出为 1 维）

4.1 线性层

线性层实现了矩阵乘法和偏置加法。在前向传播中，计算 h=xW+b，在反向传播中，计算梯度并更新权重和偏置。

```
class LinearLayer:
    def __init__(self):
        self.mem = {}

    def forward(self, x, W, b):
        h = np.matmul(x, W) + b
        self.mem={'x': x, 'W':W, 'b':b}
        return h

    def backward(self, grad_y):
        x = self.mem['x']
        W = self.mem['W']
        b = self.mem['b']
        grad_x = np.matmul(grad_y, W.T)
        grad_W = np.matmul(x.T, grad_y)
        grad_b = np.sum(grad_y, axis=0)
        return grad_x, grad_W, grad_b
```

```
class ReLU:
    def __init__(self):
        self.mem = {}

    def forward(self, x):
        self.mem['x']=x
        return np.where(x > 0, x, np.zeros_like(x))

    def backward(self, grad_y):
        x = self.mem['x']
        grad_x = np.where(x > 0,grad_y, np.zeros_like(grad_y))
        return grad_x
```

4.2 ReLU 激活函数

ReLU 激活函数在前向传播中将负值置为 0，正值保持不变。在反向传播中，梯度仅在输入大于 0 时传递。

4.3 两层 ReLU 网络

我们定义了一个两层的 ReLU 网络，包含两个线性层和一个 ReLU 激活函数。

```
class TwoLayerReLUNet:
    def __init__(self, input_size, hidden_size, output_size):
        self.W1 = np.random.normal(size=[input_size, hidden_size])
        self.W2 = np.random.normal(size=[hidden_size, output_size])
        self.b1 = np.zeros((1, hidden_size))
        self.b2 = np.zeros((1, output_size))
        self.lin1 = LinearLayer()
        self.lin2 = LinearLayer()
        self.relu = ReLU()

    def forward(self, x):
        self.h1 = self.lin1.forward(x, self.W1, self.b1)
        self.h1_relu = self.relu.forward(self.h1)
        self.h2 = self.lin2.forward(self.h1_relu, self.W2, self.b2)
        return self.h2

    def backward(self, y):
        grad_y = 2 * (self.h2 - y) / y.shape[0]
        self.h2_grad, self.W2_grad, self.b2_grad = self.lin2.backward(grad_y)
        self.h1_relu_grad = self.relu.backward(self.h2_grad)
        self.h1_grad, self.W1_grad, self.b1_grad = self.lin1.backward(self.h1_relu_grad)
```

```
def compute_loss(output, y):
    return np.mean((output - y) ** 2)

def train_one_step(model, x, y, lr):
    model.forward(x)
    model.backward(y)
    model.W1 -= lr* model.W1_grad
    model.W2 -= lr* model.W2_grad
    model.b1 -= lr* model.b1_grad
    model.b2 -= lr* model.b2_grad
    loss = compute_loss(model.h2, y)
    return loss

def test(model, x, y):
    model.forward(x)
    loss = compute_loss(model.h2, y)
    return loss
```

4.4 训练和测试

我们使用均方误差（MSE）作为损失函数，并通过梯度下降法更新模型参数。

5. 拟合效果

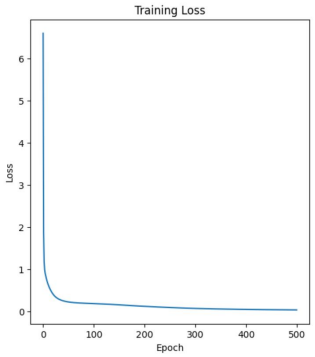
我们训练了 500 个 epoch，学习率设置为 0.01。训练过程中，每 10 个 epoch 打印一次损失值。

```
model = TwoLayerReLUNet(input_size=1, hidden_size=64, output_size=1)
losses = []
for epoch in range(500):
    loss = train_one_step(model, x_train, y_train, lr=1e-2)
    losses.append(loss)
    if epoch%10 == 0:
        print('epoch', epoch, ': loss', loss)
loss = test(model, x_test, y_test)
y_pred = model.forward(x_test)
```

5.1 训练损失

训练损失随着 epoch 的增加逐渐下降，表明模型在训练集上逐渐拟合目标函数。

```
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(losses)
plt.title("Training Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
```



5.2 拟合效果

在测试集上，模型的预测结果与真实函数非常接近，表明模型具有良好的拟合能力。

```
plt.subplot(1, 2, 2)
plt.plot(x_test, y_test, label="True Function")
plt.plot(x_test, y_pred, label="Predicted Function")
plt.title("Function Fitting")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```