函数拟合报告

一、问题重述

1. **问题描述：**使用两层 ReLU 网络拟合自定义函数
2. **理论证明：**两层 ReLU 网络的通用逼近性

一个具有足够多的隐藏单元的神经网络可以逼近任意连续函数。最早的证明主要针对 sigmoid 激活函数的神经网络，而后续研究表明，两层 ReLU 网络同样具备通用逼近能力。

二、函数定义与数据采集

1. 目标函数

$$f(x) = x^3 + x^2$$

2. 数据集生成

采用 NumPy 的 linspace 函数，在[0,1]区间均匀取样，并计算每个样本点对应的目标函数值。处理数据格式，使其适用于神经网络训练。

在 PyTorch 版本中，我们生成 100 个训练样本和 50 个测试样本，并分别存储在 x_train, y_train 和 x_test, y_test 变量中。

```
# 生成数据，num_samples为样本数量
def generate_data(num_samples=100):
    x = np.linspace(0, 1, num_samples) # 生成从0到1的等间距样本点
    y = target_function(x)  #计算目标函数的值
    return x.reshape(-1, 1), y.reshape(-1, 1) # 返回x和y的列向量形式

x_train, y_train = generate_data(100) # 生成100个训练样本
x_test, y_test = generate_data(50)  # 生成50个测试样本
```

对于 NumPy 版本,我们采用 sklearn.model_selection 提供的 train_test_split 方法将数据集划分为训练集（20%）、验证集（2.5%）和测试集（80%）。

```
x = np.linspace(0, 1, 1000) # 生成从0到1的等间距1000个样本点
y = target_function(x) # 计算目标函数的值

# 将数据集划分为训练集和测试集，测试集占80%
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.8, random_state=1)
# 将训练集进一步划分为训练集和验证集，验证集占训练集的12.5%
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.125, random_state=1)
```

三、模型描述

1. 网络架构

输入层：接收一个标量输入 x

隐藏层：包含 10（PyTorch 版本）或 128（NumPy 版本）个神经元，采用 ReLU 激活函数；

输出层：生成一个标量输出 y，用于拟合目标函数的输出。

Pytorch 版本：

```
def __init__(self):
    super(ReLUNet, self).__init__() # 初始化父类
    self.fc1 = nn.Linear(1, 10) # 定义第一个全连接层，输入1维，输出10维
    self.fc2 = nn.Linear(10, 10) # 定义第二个全连接层，输入10维，输出10维
    self.fc3 = nn.Linear(10, 1) # 定义第三个全连接层，输入10维，输出1维
    self.relu = nn.ReLU() # 定义ReLU激活函数
```

Numpy 版本：

```
input_dim = 1
hidden_dim = 128 # 隐藏层维度为128
output_dim = 1

# 初始化权重和偏置，使用He初始化方法
W1 = np.random.randn(hidden_dim, input_dim) * np.sqrt(1 / input_dim)
W2 = np.random.randn(output_dim, hidden_dim) * np.sqrt(1 / hidden_dim)
# 初始化隐藏层偏置为零
b1 = np.zeros((hidden_dim, 1))
b2 = np.zeros((output_dim, 1))

cache = {}

 # 线性变换函数
def linear(W, X, b):
    return np.matmul(W, X) + b
```

2. 激活函数

模型采用 ReLU 作为隐藏层的激活函数

Pytorch 版本：

```
# 前向传播过程
def forward(self, x):
    x = self.relu(self.fc1(x)) # 第一层全连接层后接ReLU激活函数
    x = self.relu(self.fc2(x)) # 第二层全连接层后接ReLU激活函数
    x = self.fc3(x)  # 第三层全连接层，输出预测值
    return x
```

Numpy 版本：

```
# ReLU激活函数
def relu(X):
    return np.where(X > 0, X, 0)

# 前向传播函数
def forward(x):
    x = x.reshape(1, -1) # 将输入x转换为列向量
    cache['out0'] = x # 存储输入x到缓存
    x = linear(W1, x, b1) # 第一层线性变换
    cache['out1'] = x # 存储第一层输出到缓存
    x = relu(x) # 第一层ReLU激活
    cache['out2'] = x # 存储ReLU输出到缓存
    output = linear(W2, x, b2) # 第二层线性变换
    cache['out3'] = output # 存储第二层输出到缓存
    return output # 返回最终输出
```

3. 损失函数

为了衡量神经网络的拟合效果，使用均方误差量化模型预测值与真实值之间的偏差。

Pytorch 版本：

```
criterion = nn.MSELoss() # 定义损失函数为均方误差损失
```

Numpy 版本：

```
# 损失函数，计算均方误差
def loss_fn(y_pred, target):
    loss = np.sum((y_pred - target) ** 2, axis=1, keepdims=True) / len(y_pred)
    cache['loss'] = (y_pred - target) / len(y_pred) # 存储损失的梯度到缓存
    return loss
```

4. 优化方法

PyTorch 版本使用 Adam 作为优化算法。

```
optimizer = optim.Adam(model.parameters(), lr=0.01) # 定义优化器为Adam，学习率为0.01
```

NumPy 版本实现了基于梯度下降的手动更新机制，通过反向传播计算梯度并更新参数。

```
# 反向传播函数
def backward(lr, batch_size):
    global W1, W2, b1, b2

    delta2 = cache['loss'] / batch_size # 计算第二层损失的梯度
    dW2 = np.matmul(delta2, cache['out2'].T) / batch_size # 计算W2的梯度
    db2 = np.sum(delta2, axis=1, keepdims=True) / batch_size # 计算b2的梯度

    delta1 = np.matmul(W2.T, delta2) * (cache['out1'] > 0) # 计算第一层损失的梯度
    dW1 = np.matmul(delta1, cache['out0'].T) / batch_size # 计算W1的梯度
    db1 = np.sum(delta1, axis=1, keepdims=True) / batch_size # 计算b1的梯度

    # 更新权重和偏置
    W1 -= lr * dW1
    W2 -= lr * dW2
    b1 -= lr * db1
    b2 -= lr * db2
```

5. 训练过程（超参数、训练策略）

PyTorch 版本：

训练模型时，使用 train_model()函数进行 3000 轮训练，每 100 轮打印一次损失。

```python
for epoch in range(3000): # 训练3000个epoch
    optimizer.zero_grad() # 清零梯度
    output = model(x_train_tensor) # 前向传播，得到预测值
    loss = criterion(output, y_train_tensor) # 计算损失
    loss.backward() # 反向传播，计算梯度
    optimizer.step() # 更新模型参数

    if epoch % 100 == 0: # 每100个epoch打印一次损失
        print(f'Epoch {epoch}, Loss: {loss.item()}') # 打印当前epoch和损失值
```

NumPy 版本:

训练过程采用手动计算梯度更新参数，使用小批量 SGD 进行梯度下降，每次从训练集中随机取 32 个样本进行参数更新，设置 100000 轮训练，并每 10000 轮打印一次损失。

```python
# 训练模型
def train(X, y, epochs=100000, batch_size=32, lr=0.01, verbose_ep=10000):
    for epoch in range(epochs):
        rec_loss = 0
        for batch_idx in range(len(X) // batch_size): # 按批次处理数据
            # 获取当前批次的输入数据和标签数据
            X_batched = np.array([X[batch_idx * batch_size: min((batch_idx + 1) * batch_size, len(X))]])
            y_batched = np.array([y[batch_idx * batch_size: min((batch_idx + 1) * batch_size, len(y))]])
            y_pred = forward(X_batched) # 前向传播，得到预测值
            # 计算损失
            loss = np.sum(loss_fn(y_pred, y_batched)) / len(y_pred)
            rec_loss += loss # 累加损失
            backward(lr, batch_size) # 反向传播，更新参数
        if epoch % verbose_ep == 0: # 打印epoch和平均损失
            print(f"Epoch:{epoch}, Loss:{rec_loss / (len(X) // batch_size)}")
```

四、拟合效果

1. 预测结果可视化

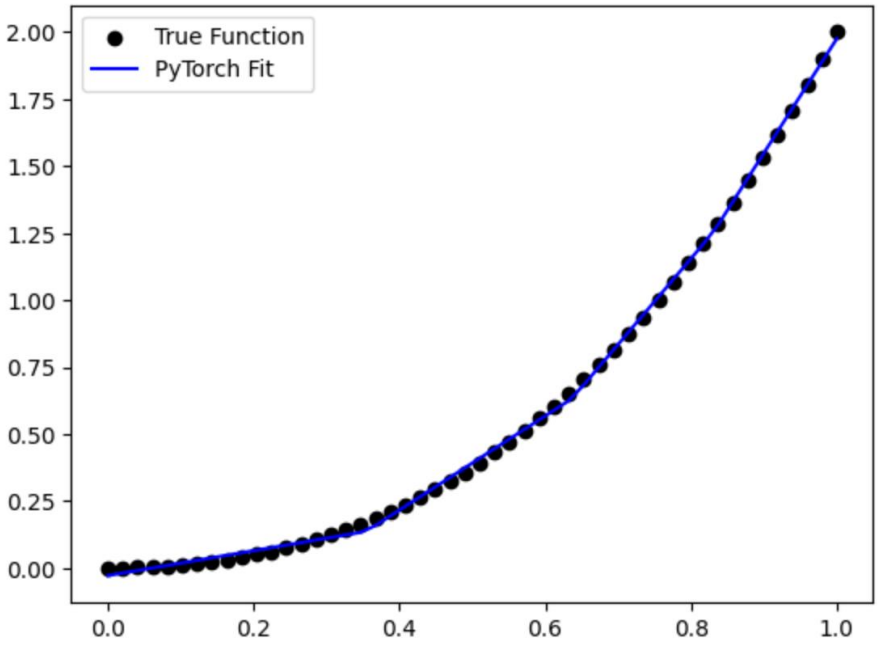
```python
# 可视化拟合结果
def visualize_results():
    model_pytorch = train_model() # 训练模型并获取模型实例

    x_test_tensor = torch.tensor(x_test, dtype=torch.float32) # 将测试数据转换为PyTorch张量
    y_pred_pytorch = model_pytorch(x_test_tensor).detach().numpy() # 使用模型进行预测，并将预测结果转换为NumPy

    plt.scatter(x_test, y_test, label='True Function', color='black') # 绘制真实函数的散点图
    plt.plot(x_test, y_pred_pytorch, label='PyTorch Fit', color='blue') # 绘制模型拟合的曲线
    plt.legend() # 显示图例
    plt.show() # 显示图形
```

PyTorch 版本：

训练的两层 ReLU 神经网络 能够很好地拟合曲线，预测值（蓝色曲线）与真实函数值（黑色点）基本一致。由于使用 Adam 优化器，训练收敛较快，拟合效果较优。



Numpy 版本：

NumPy 版本的神经网络也能够成功拟合目标函数，由于使用手动实现的随机梯度下降，训练较慢，损失下降速度低于 PyTorch 版本。