Sin(x)函数近似与 ReLU 神经网络

一、引言

在本报告中，我们实现了一个具有 ReLU 激活函数的两层神经网络，用于近似函数 Sin(x)。该网络使用梯度下降法并结合动态学习率调度进行训练。我们将通过在合成数据上训练并比较预测值与真实函数，来展示该网络的有效性。

二、方法

该神经网络基于一个两层架构：

- **输入层：** 一个输入神经元，用于接收 x 值。
- **隐藏层：** 150 个神经元，使用 ReLU 激活函数。
- **输出层：** 一个输出神经元，表示预测的 y 值。

1. 需要近似的函数

目标函数定义为：f(x) = Sin(x)

2. 网络结构

网络的实现如下：

- 第一层的权重（W_1）使用 He 初始化方法进行初始化，这对于 ReLU 激活函数是合适的。
- 第二层（W_2）将隐藏层与输出层连接。
- 隐藏层应用 ReLU 激活函数，这意味着隐藏层中的负值会被置为零。

3. 学习率调度

学习率在训练过程中动态衰减，由以下公式控制：

$$\text{learning rate} = \text{initial learning rate} \times (\text{decay rate})^{\frac{\text{epoch}}{\text{decay steps}}}$$

这有助于模型在训练过程中更加有效地收敛。

三、代码实现

1. 神经网络类

```
    class ReLU_Network:
        def __init__(self):
            self.W1 = np.random.randn(1, 150).astype(np.float32) *
np.sqrt(2/1)
            self.b1 = np.zeros(150, dtype=np.float32)
            self.W2 = np.random.randn(150, 1).astype(np.float32) *
np.sqrt(2/100)
            self.b2 = np.zeros(1, dtype=np.float32)

        def forward(self, x):
            hidden_preactivation = np.dot(x, self.W1) + self.b1
            hidden_layer = np.maximum(0, hidden_preactivation)  #
ReLU 激活
            output = np.dot(hidden_layer, self.W2) + self.b2
            return output, hidden_layer, hidden_preactivation
```

2. 训练函数

```
    def   train_model(model,   x_train,   y_train,   epochs=2000,
initial_learning_rate=0.001):
        losses = []
        for epoch in range(epochs):
            learning_rate                                          =
learning_rate_schedule(initial_learning_rate, epoch)
            output,    hidden_layer,    hidden_preactivation    =
model.forward(x_train)
            loss = np.mean((output - y_train)**2)
            losses.append(loss)

            batch_size = x_train.shape[0]
            d_output = 2 * (output - y_train) / batch_size
            dW2 = np.dot(hidden_layer.T, d_output)
            db2 = np.sum(d_output, axis=0)

            d_hidden = np.dot(d_output, model.W2.T)
            d_preactivation = d_hidden * (hidden_preactivation > 0)
```

```
            dW1 = np.dot(x_train.T, d_preactivation)
            db1 = np.sum(d_preactivation, axis=0)

            model.W1 -= learning_rate * dW1
            model.b1 -= learning_rate * db1
            model.W2 -= learning_rate * dW2
            model.b2 -= learning_rate * db2
```

3. 测试函数

```
def test_model(model, x_test, y_test):
    output, _, _ = model.forward(x_test)
    test_loss = np.mean((output - y_test)**2)
     print(f"Test Loss: {test_loss:.4f}")
```

四、结果

1. 训练过程

在训练过程中，网络的损失函数逐渐减小，随着模型学习到对函数 Sin(x)的近似。学习率衰减帮助模型在训练过程中平稳地收敛。

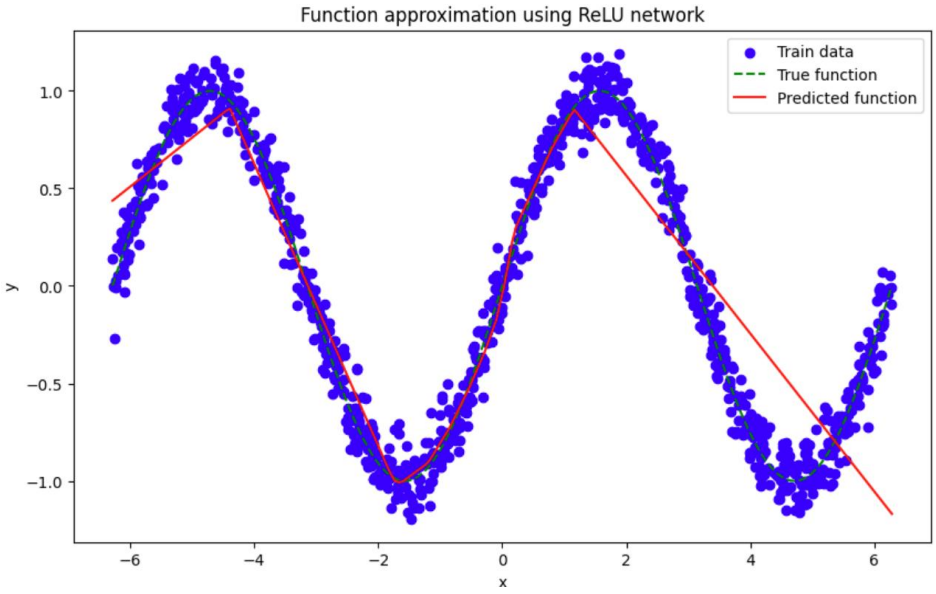
2. 测试损失

训练结束后，我们在单独的测试集上测试模型的表现。测试损失通过计算预测输出与真实函数之间的均方误差来获得。这能反映出模型对未见数据的泛化能力。

3. 可视化

以下图表显示了真实函数和神经网络预测函数之间的比较：

- **蓝色点：** 训练数据点（带噪声）。
- **绿色虚线：** 真实函数 Sin(x)
- **红色线：** 神经网络预测的函数。



五、小结

在本报告中，我们通过实现一个两层 ReLU 神经网络，成功地近似了函数 Sin(x)。该网络使用了梯度下降法与动态学习率调度相结合的方式进行训练，有效地优化了模型的表现。

1. 主要发现：

1）网络结构与训练过程： 我们采用了一个简单的两层网络，其中第一层使用 ReLU 激活函数，第二层为输出层。通过梯度下降法，结合动态学习率调度，网络在训练过程中逐渐学习到函数的近似。

2）学习率调度的作用： 学习率的动态衰减帮助模型更好地收敛，避免了在训练后期出现震荡或过早收敛的问题。

3）可视化与性能： 通过可视化结果，网络的预测值与真实函数 Sin(x)之间的差异逐步减小，表明网络能够有效地学习目标函数。

2. 不使用 TensorFlow 的原因：

本项目中，我们没有使用 TensorFlow 或其他深度学习框架，而是使用了原生的 NumPy 来实现神经网络。这是因为本实验的目的是展示基本的神经网络原理和训练过程，而使用 NumPy 能够帮助我们更清晰地理解网络的工作原理和计算过程。