神经网络第三次作业

201300086 史浩男

一二、感知机

1. 什么是单层感知机的局限性，造成其局限性的原因是什么，该如何解决？
2. 相比单层感知机，多层感知机有什么优点，又带来什么问题？

单层感知机局限性：只能解决线性可分问题。

单层感知机局限性原因：输出层只有一个，并且使用线性激活函数，这导致决策边界只能是一个线性函数

解决单层感知机局限性：使用多层感知机MLP

MLP优点：

- 处理非线性问题：MLP具有多个隐藏层，每个隐藏层可以使用非线性激活函数，可以处理非线性可分的问题。
- 更强大的模型表示能力：多个隐藏层使MLP可以学习更复杂的函数
- 避免局部最优解：MLP使用更先进的优化算法，如Adam

MLP带来的问题：

- 过拟合，可以使用正则化、早停等技术来避免
- MLP具有多个隐藏层与大量参数，计算资源要求高
- 调参难度增大

三、误差函数

3. 比较绝对误差与平方误差的优劣，并介绍其他的任意两种误差函数。

绝对误差：

$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$。

- 对异常值的敏感度较低。
- 由于它的导数在零点处不连续，这在某些优化算法（如梯度下降）中可能会造成问题。

平方误差：

$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$。

- 对大的误差惩罚更大，因此更加重视那些有很大偏差的预测。
- 导数在所有位置都是连续的，这使得它在优化算法中更方便使用。
- 对于异常值的敏感度较高，这可能会在存在异常值的情况下导致模型的性能下降。

对数损失函数（Log Loss）：对数损失函数通常用于二元或多元分类问题，它是对数似然损失函数的一种特殊情况。对数损失函数对于预测的概率分布和实际的概率分布之间的差距进行度量，对预测错误的类别的概率较低的模型进行较大的惩罚。

交叉熵损失函数（Cross-Entropy Loss）：度量实际分布与预测分布之间的差距。它的优点是在训练深度学习模型时，能够有效地避免梯度消失的问题，从而使模型学习更有效。

四、

4. 描述梯度下降的优缺点和局部最小值的定义，使用梯度下降方法优化 Himmelblau 函数：$f(x,y)=(x^2+y-11)^2+(x+y^2-7)^2$。可使用 Pytorch 框架。（Himmelblau 函数可视化代码如下）

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
def him(x):
    return (x[0]**2+x[1]-11)**2 + (x[0]+x[1]**2-7)**2

x = np.arange(-6,6,0.1)
y = np.arange(-6,6,0.1)
X,Y = np.meshgrid(x,y)
Z = him([X,Y])

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(X,Y,Z,cmap='rainbow')
ax.set_xlabel('x[0]')
ax.set_ylabel('x[1]')
ax.set_zlabel('f')
fig.show()
```

局部最小值：损失函数中的某个点，其周围所有方向的斜率都为零或近似为零，但该点并不是全局最小值点。

梯度下降优点：简单易于实现，能够高效地求解模型参数，可以被扩展到各种各样的神经网络结构中

梯度下降缺点：可能陷入局部最小值，尤其是损失函数非凸的时候，容易无法找到全局最小值。而且依赖合适学习率，过高过低都可能影响算法性能

增加的代码如下：

```
x = torch.tensor([0., 0.], requires_grad=True)#初始化需要优化的参数
optimizer = torch.optim.Adam([x], lr=1e-2) # 学习率

for step in range(1000):
    pred = him(x)
    optimizer.zero_grad()
    pred.backward()
    optimizer.step()
    if (step+1) % 100 == 0:#每100epoch输出
        print(f"step={step+1}, x={x.tolist()}, f(x)={pred.item()}")
```

五、

代码设计单层感知机，使其满足 $y = \sigma(W x + b)$。（其函数定义如以下代码，只需完成其实现）

```
import numpy as np


class Layer():
    def __init__(self, input_dim, output_dim, bias=True):
        # 初始化权重矩阵和偏置项
        self.weights = np.random.normal(loc=0.0, scale=np.sqrt(
            2 / (input_dim + output_dim)), size=(input_dim, output_dim))
        if bias:
            self.bias = np.zeros((1, output_dim))
        else:
            self.bias = None

    def __call__(self, x, train=False):
        # 前向传播计算
        return self.forward(x, train)

    def forward(self, x, train):
        # 前向传播计算
        self.input = x
        self.output = np.dot(self.input, self.weights)
        if self.bias is not None:
            self.output += self.bias
        return self.output

    def backward(self, error, eta):
```

```
        # 反向传播计算
        if self.bias is not None:
            self.gradient_bias = np.sum(error, axis=0, keepdims=True)
        self.gradient_weights = np.dot(self.input.T, error)
        error = np.dot(error, self.weights.T)
        # 更新权重矩阵和偏置项
        self.weights -= eta * self.gradient_weights
        if self.bias is not None:
            self.bias -= eta * self.gradient_bias
        return error
```

六、

在第 5 题的基础上，使用 Layer 类完成多层感知机的构造，其成员函数和 Layer 一致，并用于对 $\cos(2\pi x)$ 函数的拟合。

```
class MLP():
    def __init__(self, input_dim, hidden_dim, output_dim, hidden_layers, bias=True):
        # 初始化神经网络结构
        self.layers = []
        self.layers.append(Layer(input_dim, hidden_dim, bias))
        for i in range(hidden_layers):
            self.layers.append(Layer(hidden_dim, hidden_dim, bias))
        self.layers.append(Layer(hidden_dim, output_dim, bias))
        self.activation = lambda x: np.where(x > 0, x, x * 0.1)

        # 初始化Adam优化器的参数
        self.m_weights = []
        self.v_weights = []
        self.m_bias = []
        self.v_bias = []
        for layer in self.layers:
            self.m_weights.append(np.zeros_like(layer.weights))
            self.v_weights.append(np.zeros_like(layer.weights))
            if layer.bias is not None:
                self.m_bias.append(np.zeros_like(layer.bias))
                self.v_bias.append(np.zeros_like(layer.bias))

    def forward(self, x, train):
        for layer in self.layers:
            x = self.activation(layer.forward(x, train))
        return x

    def backward(self, error, eta, beta1=0.8, beta2=0.9, epsilon=1e-8):
        for i in reversed(range(len(self.layers))):
            layer = self.layers[i]
            error = layer.backward(error, eta)
```

```
            # 更新m和v
            self.m_weights[i] = beta1 * self.m_weights[i] + (1 - beta1) *
layer.gradient_weights
            self.v_weights[i] = beta2 * self.v_weights[i] + (1 - beta2) *
layer.gradient_weights ** 2
            m_hat_weights = self.m_weights[i] / (1 - beta1)
            v_hat_weights = self.v_weights[i] / (1 - beta2)

            # 使用Adam的更新规则更新权重
            layer.weights -= eta * m_hat_weights / (np.sqrt(v_hat_weights) +
epsilon)

            if layer.bias is not None:
                self.m_bias[i] = beta1 * self.m_bias[i] + (1 - beta1) *
layer.gradient_bias
                self.v_bias[i] = beta2 * self.v_bias[i] + (1 - beta2) *
layer.gradient_bias ** 2
                m_hat_bias = self.m_bias[i] / (1 - beta1)
                v_hat_bias = self.v_bias[i] / (1 - beta2)

                # 使用Adam的更新规则更新偏置
                layer.bias -= eta * m_hat_bias / (np.sqrt(v_hat_bias) + epsilon)

        return error

    def train(self, x, y, epochs, eta, batch_size):
        # 训练神经网络
        n_samples = x.shape[0]
        for epoch in range(epochs):
            indices = np.random.permutation(n_samples)
            for start_idx in range(0, n_samples, batch_size):
                end_idx = min(start_idx + batch_size, n_samples)
                batch_indices = indices[start_idx:end_idx]
                x_batch = x[batch_indices]
                y_batch = y[batch_indices]
                output = self.forward(x_batch, train=True)
                error = output - y_batch
                self.backward(error, eta)

    def test(self, x):
        output = self.forward(x, train=False)
        return output

if __name__ == '__main__':
    # 生成训练数据
    x = np.linspace(0, 1, 1000)
    y = np.cos(2 * np.pi * x)

    # 创建、训练、测试神经网络
    net = MLP(input_dim=1, hidden_dim=10, output_dim=1, hidden_layers=5)
    net.train(x[:, np.newaxis], y[:, np.newaxis], epochs=100, eta=0.01,
batch_size=10)
```

```
    y_pred = net.test(x[:, np.newaxis])



    #每隔100epoch输出损失
    for epoch in range(100):
        if epoch % 10 == 0:
            output = net.test(x[:, np.newaxis])
            loss = np.mean((output - y[:, np.newaxis]) ** 2)
            print('epoch: {}, loss: {}'.format(epoch, loss))


    # 可视化
    plt.plot(x, y, label='true')
    plt.plot(x, y_pred, label='pred')
    plt.legend()
    plt.show()
```