作业

Homework3

庄镇华 502022370071

A Neural Networks Homework Assignment

NANJING UNIVERSITY

2023 年 5 月 15 日

注意：对于 4、5、6 题，请在 pdf 文件中贴代码，并配以必要的注释。

题目一

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？

解答： 单层感知机的局限指其无法解决线性不可分问题，比如异或问题。对于非线性问题，需要增加神经元，以拟合输入-输出关系。

解决方法是增加网络层数，即增加隐藏层，这样单层感知机就变成了多层感知机，不仅可以解决异或问题，而且具有很好的非线性分类效果。

题目二

相比单层感知机，多层感知机有什么优点，又带来什么问题？

解答： 多层感知机的优点有：

1. **更好地解决非线性问题**。多层感知机可以通过训练权值参数来学习非线性函数，这对于解决很多实际问题是很有用的。
2. **具有较高的准确率**。多层感知机可以通过训练来获得较高的准确率，尤其是当数据集较大时。
3. **具有较好的泛化能力**。多层感知机可以很好地适应新的数据，对于未来的数据具有较好的泛化能力。
4. **训练方式灵活**。多层感知机可以使用不同的优化算法和损失函数来训练，这使得它可以应用于各种不同的场景。

随着层数的增多带来的问题有：

1. **过拟合**，可通过 DropOut 解决。
2. **参数难以调试**，可通过 Adagrad、Adam、Adadelta 等自适应的梯度下降方法降低调试参数的负担。
3. **梯度弥散**。使用 Sigmoid 在反向传播中梯度值会逐渐减少，经过多层的传递后会呈指数级的剧烈减少，因此梯度值在传递到前面几层时就变得非常小了，神经网络参数的更新将会非常缓慢。

题目三

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。

解答： 绝对误差函数曲线呈 V 字型，在 0 处不可导，计算机求解导数比较困难，而且该函数大部分情况下梯度都是相等的，这意味着即使对于小的损失值，其梯度也是大的，不利于模型的收敛和学习。但绝对误差对离群点不那么敏感，更有包容性，因为该函数计算的是误差的绝对值，无论是误差大于 1 还是小于 1，没有平方项的作用，惩罚力度都是一样的。

平方误差处处可导，计算机求解梯度较为容易，且其梯度也根据误差值而动态变化，能较快准确达到收敛。但是从离群点角度来看，平方误差很容易受到异常点的影响，对异常点会赋予较大的权重，如果异常点不属于考虑范围，是由于某种错误导致的，则此函数指导方向将出现偏差。

其他的误差函数：

Huber 损失

$$L_\delta(y,f(x)) = \begin{cases} \frac{1}{2}(y-f(x))^2 & \text{for } |y-f(x)| \le \delta, \\ \delta|y-f(x)| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$

优点：对异常值更加鲁棒；在最优点附近由于调整为 MSE，梯度更新会随着误差减小而减小，有利于收敛。缺点：引入额外的超参，需要调试；临界点处不可导。

Log-Cosh 损失

$$L(y,y^p) = \sum_{i=1}^{n} \log(\cosh(y_i^p - y_i))$$

优点：具有 huber 损失具备的所有优点；二阶处处可微，许多机器学习算法比如 XGBoost 算法采用牛顿法逼近最优点，而牛顿法要求损失函数二阶可微。缺点：误差很大情况下，一阶梯度和 Hessian 会变成定值，导致 XGBoost 出现缺少分裂点的情况。

题目四

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()
```

解答：　梯度下降的优势

1. 相比大规模数值矩阵，梯度下降算法的迭代求解效率更高。
2. 对于无法计算全域唯一优解的情况，梯度下降仍然能够有效进行最小值点求解。

梯度下降的缺点

1. 局部最小陷阱：指的是该点左右两端取值都大于该点，但是该点不是全域最小值点。

2. 鞍点陷阱：鞍点是指那些不是极值点但梯度为 0 的点

3. 当网络层数特别深时，会存在梯度爆炸和梯度消失现象。

局部极小值：如果存在一个 $\epsilon > 0$，使的所有满足 $|x - x^{\star}| < \epsilon$ 的 x 都有 $f(x^{\star}) \le f(x)$ 我们就把点 $x^{\star}$ 对应的函数值 $f(x^{\star})$ 称为一个函数 f 的局部最小值。

```python
import torch
lr = 1e-3 # 学习率
T = 20000 # 迭代次数
if __name__ == '__main__':
    # x代表坐标值(x,y)
    x = torch.tensor([0., 0.], requires_grad=True)
    # 定义Adam优化器, 学习速率是1e-3
    optimizer = torch.optim.Adam([x], lr=lr)
    for step in range(T):
        # 输入坐标, 得到预测值
        pred = him(x)
        # 梯度清零
        optimizer.zero_grad()
        # 梯度回传, 获取坐标的梯度信息
        pred.backward()
        # 沿梯度方向更新梯度, 优化坐标值
        optimizer.step()

        if step % (T // 10) == 0:
            print('step {}: x = {}, f(x) = {}'.format(step, x.tolist(), pred.item
    ()))

# 输出情况
'''
step 0:     x = [0.0009999999310821295, 0.0009999999310821295], f(x) = 170.0
step 2000:  x = [2.3331809043884277, 1.9540693759918213], f(x) =
    13.730910301208496
step 4000:  x = [2.9820079803466797, 2.0270984172821045], f(x) =
    0.014858869835734367
step 6000:  x = [2.999983549118042, 2.0000221729278564], f(x) = 1.1074007488787174
    e-08
step 8000:  x = [2.9999938011169434, 2.0000083446502686], f(x) =
    1.5572823031106964e-09
step 10000: x = [2.999997854232788, 2.000002861022949], f(x) = 1.8189894035458565e
    -10
step 12000: x = [2.9999992847442627, 2.0000009536743164], f(x) =
    1.6370904631912708e-11
step 14000: x = [2.999999761581421, 2.000000238418579], f(x) = 1.8189894035458565e
    -12
step 16000: x = [3.0, 2.0], f(x) = 0.0
step 18000: x = [3.0, 2.0], f(x) = 0.0
```

函数图像可视化如下图所示：

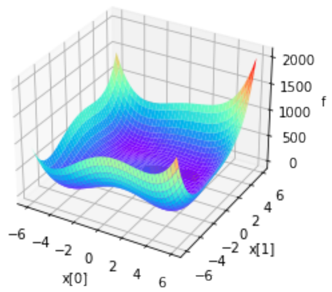

图 1: Himmelbalu 函数可视化

题目五

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

```
class Layer():
    def __init__(self,input_dim,outpu_dim,bias=True):
        None
    def __call__(self,x,train=False):
        None
    def forward(self,x,train):
        None
    # 反向传播函数
    def backward(self,error,eta):
        None
```

解答： 单层感知机实现代码如下，具体实现细节见注释。

```
import numpy as np

# 激活函数
def relu(x):
    return np.maximum(x, 0)

# 激活函数导数
def relu_derivative(x):
    grad = np.array(x, copy=True)
    grad[x >  0] = 1.
    grad[x <= 0] = 0.
    return grad

class Layer():
    def __init__(self,input_dim,output_dim,bias=True):
        super(Layer, self).__init__()
        # 输入维度
        self.input_dim = input_dim
        # 输出维度
        self.output_dim = output_dim
        # 随机初始化权重
        self.w = np.random.randn(self.input_dim, self.output_dim)
```

```python
        self.bias = bias
        # 初始化激活函数
        self.activation = relu
        self.activation_derivative = relu_derivative
        # 随机初始化偏置
        if self.bias:
            self.b = np.random.randn(self.output_dim)

    def __call__(self,x,train=False):
        # __call__函数使得类对象具有类似函数的功能
        # 直接内部调用forward函数即可
        z = self.forward(x,train)
        return z

    def forward(self,x,train):
        # 计算wx + b
        y = np.dot(x, self.w) + self.b
        # 计算\sigma(wx + b)
        z = self.activation(y)
        # 如果是训练模式，则需要计算梯度
        if train:
            self.pre_output = x
            self.grad_z = self.activation_derivative(y)
        return z

    # 反向传播函数
    def backward(self,error,eta):
        '''
        反向传播过程 error->out->net->w,b
        '''
        # d_out_net代表out对net的求导结果
        d_out_net = self.grad_z
        # d_net_w代表net对w的求导结果
        d_net_w = self.pre_output


        # d_error_net代表error对net的求导结果，这里应用了链式法则
        d_error_net = np.multiply(d_out_net, error)

        # d_error_w代表error对w的求导结果，这里应用了链式法则
        d_error_w = np.dot(d_net_w.T, d_error_net)

        self.dw = d_error_w

        # 计算偏置的权重
        if self.bias:
            d_error_b = d_error_net.sum(axis=0)
            self.db = d_error_b

        # 给下一层的error对out的求导的结果为上一层的加权和
        self.d_error_out_ = np.dot(d_error_net, self.w.T)

        # eta为学习率，进行梯度更新
        self.w -= eta * self.dw
```

```
        self.b -= eta * self.db

        return self.d_error_out_
```

测试代码如下所示，经测试可以收敛

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split

x, y = datasets.make_moons(n_samples=1000, noise=0.2, random_state=100)
x, y = x.reshape(1000, 2), y.reshape(1000, 1)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3,
    random_state=42)
print(x.shape, y.shape)

np.random.seed(2023)
layer = Layer(2, 1, True)

for i in range(800):
    y_pred = layer.forward(x_train, True)
    error = y_pred - y_train
    layer.backward(error, 0.0001)
    if i % 80 == 0:
        mse = np.mean(np.square(error))
        print(mse)
```

题目六

在题目五的基础上，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。

解答： 多层感知机实现代码如下，具体实现细节见注释。

```
def swish(x):
    beta = 0.2
    return x * (1 / (1 + np.exp(-beta * x)))

def swish_derivative(x):
    beta = 0.2
    sigma = (1 / (1 + np.exp(-beta * x)))
    fx = x * sigma
    return beta * fx + sigma * (1 - beta * fx)

# 多层感知机
class MultiLayer():
    def __init__(self,layer_dim_list,lr,bias=True):
        super(MultiLayer, self).__init__()
        # 单层感知机列表
        self.layer_list = []
        # 学习率
        self.lr = lr
        for i in range(len(layer_dim_list)-1):
            input_dim = layer_dim_list[i]
```

```python
            output_dim = layer_dim_list[i+1]
            self.layer_list.append(Layer(input_dim,output_dim,bias))
        # 多层感知机层数
        self.layer_num = len(self.layer_list)

    def __call__(self,x,train=False):
        # __call__函数使得类对象具有类似函数的功能
        # 直接内部调用forward函数即可
        z = self.forward(x,train)
        return z

    def forward(self,x,train):
        x = x.reshape(x.shape[0], -1)
        out = x
        for layer in self.layer_list:
            out = layer.forward(x,True)
            x = out
        return out

    # 反向传播函数
    def backward(self,error,eta):
        # 从后往前进行梯度更新
        for idx in range(self.layer_num-1,-1,-1):
            error = self.layer_list[idx].backward(error,eta)
        return None
```

经过实验验证，当激活函数为 swish 函数时，对 $\cos(2\pi x)$ 函数的拟合效果较好。测试代码如下所示，经测试可以收敛

```python
x = np.linspace(-0.5, 0.6, 1000, endpoint=True)
y = np.cos(2*np.pi*x)
x_train, y_train = x.reshape(x.shape[0], 1), y.reshape(y.shape[0], 1)

np.random.seed(2028)
layer = MultiLayer([1,8,1], 0.0004, True)
print(layer.layer_num)

for i in range(80000):
    y_pred = layer.forward(x_train, True)
    error = y_pred - y_train
    layer.backward(error, 0.0004)
    if i % 800 == 0:
        mse = np.mean(np.square(error))
        print(mse)
```

拟合效果如下图所示

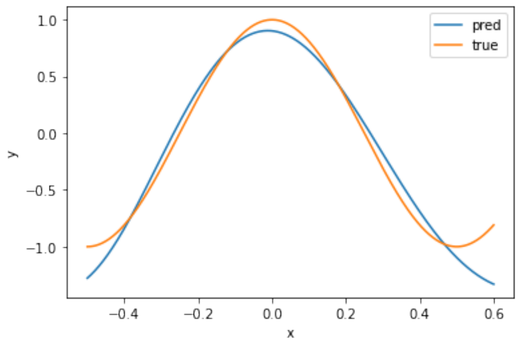

图 2: 余弦函数拟合效果