L1 AND L2 REGULARIZATION

一、内容

L1正则化，由于其线性特性，比L2正则化更多地惩罚小权重，导致模型开始对小输入不敏感，只对较大的输入变化。这就是为什么L1正则化很少单独使用，通常如果使用的话，也会与L2正则化结合。这种类型的正则化函数使权重和参数的和趋向于0，这也可以帮助解决梯度爆炸（模型不稳定，可能导致权重变成非常大的值）的情况。

二、前向传播

一、公式

$$L_{1w} = \lambda \sum_{i=k} |w_k|$$

$$L_{1b} = \lambda \sum_{i=k} |b_k|$$

$$L_{2w} = \lambda \sum_{i=k} w_k^2$$

$$L_{2b} = \lambda \sum_{i=k} b_k^2$$

$$Loss = dataloss + L_{1w} + L_{1b} + L_{2w} + L_{2b}$$

二、实现

```
class Layer_Dense:
    def __init__(self, n_input, n_neuron, weight_L1, weight_L2, bias_L1, bias_L2):
        # 用正态分布初始化权重
        self.weight = 0.01 * np.random.randn(n_input, n_neuron)
        # 将bias(偏差)初始化为0
        # self.bias = np.zeros(n_neuron)
        self.bias = np.zeros((1, n_neuron))
        self.weight_L1 = weight_L1
        self.weight_L2 = weight_L2
        self.bias_L1 = bias_L1
        self.bias_L2 = bias_L2
```

> 因为weight_L1, weight_L2, bias_L1, bias_L2和weight、bias是同时使用，所以以属性值存在Layer_Dense中。

```
class Loss:
    def regularization_loss(self,layer):
        # 默认为0
        regularization_loss = 0
        # 如果存在L1的loss
        if layer.weight_L1 > 0:
            regularization_loss += layer.weight_L1 * np.sum(np.abs(layer.weight))
        if layer.bias_L1 > 0:
            regularization_loss += layer.bias_L1 * np.sum(np.abs(layer.bias))
        # 如果存在L2的loss
        if layer.weight_L2 > 0:
            regularization_loss += layer.weight_L2 * np.sum(layer.weight ** 2)
        if layer.bias_L2 > 0:
            regularization_loss += layer.bias_L2 * np.sum(layer.bias ** 2)

        return regularization_loss
```

> Loss类中要有反回regularization_loss的方法

三、反向传播

一、公式

$$L_{2w} = \lambda \sum_m w_m^2 \quad \rightarrow \quad \frac{\partial L_{2w}}{\partial w_m} = \frac{\partial}{\partial w_m}[\lambda \sum_m w_m^2] =$$

$$= \lambda \frac{\partial}{\partial w_m} w_m^2 = \lambda \cdot 2w_m^{2-1} = 2\lambda w_m$$

$$L_{1w} = \lambda \sum_m |w_m| \quad \rightarrow \quad L'_{1w} = \frac{\partial}{\partial w_m} \lambda \sum_m |w_m| = \lambda \frac{\partial}{\partial w_m} |w_m| = \lambda \begin{cases} 1 & w_m > 0 \\ -1 & w_m < 0 \end{cases}$$

二、实现

```
class Layer_Dense:
    def backward(self, dvalue):
        # dvalue是loss对下一层（Activation）的输入（input）的导数，
        # 也就是loss对这一层（Layer_Dense）的输出（output）的导数，
        # 这里会用到链式法则

        # 在本层中，希望求得的是loss对这一层（Layer_Dense）的self.weight的导数
        # 这便找到了self.weight优化的方向（negative gradient direction）

        # 这里要考虑到self.dweight的大小要与self.weight一致，因为方便w - lr * dw公式进
行优化
        # 假设input只有一个sample，大小为1xa，weight大小为axb，则output大小为1xb，
        # 因为loss是标量，所以dvalue = dloss/doutput大小即为output的大小(1xb)，
        # 所以dweight的大小为(1xa).T * (1xb) = axb,大小和weight一致。
        # 注意：当input有多个sample时（一个矩阵输入），则dweight为多个axb矩阵相加。
        self.dweight = np.dot(self.input.T, dvalue)

        # 在本层中，希望求得的是loss对这一层（Layer_Dense）的self.input的导数
        # 以便作为下一层的backward方法中的dvalue参数，

        # 因为loss是标量，所以dinput大小即为intput的大小(1xa)，
        # dvalue = dloss/doutput大小即为output的大小(1xb)，
        # weight大小为axb
        # 所以1xa = (1xb) * (axb).T
        self.dinput = np.dot(dvalue, self.weight.T)
```

```
        # 像self.dinput一样，self.dbias可以通过矩阵乘法实现，
        # self.dbias = np.dot( dvalue, np.ones( ( len(self.bias), len(self.bias) ) ) )
        # 但有更快更简单的实现
        self.dbias = np.sum(dvalue, axis=0, keepdims=True)   # 此处不要keepdims=True也
行，因为按0维相加还是行向量

        # 正则项的梯度
        if self.weight_L2 > 0:
            self.dweight += 2 * self.weight_L2 * self.weight
        if self.bias_L2 > 0:
            self.dbias += 2 * self.bias_L2 * self.weight
        if self.weight_L1 > 0:
            dL = np.ones_like(self.weight)
            dL[self.weight < 0] = -1
            self.dweight += self.weight_L1 * dL
        if self.bias_L1 > 0:
            dL = np.ones_like(self.bias)
            dL[self.bias < 0] = -1
            self.dbias += self.bias_L1 * dL
```

三、实例

```
# 数据集
X, y = spiral_data(samples=2000, classes=3)
keys = np.array(range(X.shape[0]))
np.random.shuffle(keys)
X = X[keys]
y = y[keys]
X_test = X[3000:]
y_test = y[3000:]
X = X[0:3000]
y = y[0:3000]
print(X-X_test)

# 2输入64输出
dense1 = Layer_Dense(2, 512, weight_L2=5e-4, bias_L2=5e-4)#, weight_L2=5e-4, bias_L2=5e-4
activation1 = Activation_ReLu()
# 64输入3输出
dense2 = Layer_Dense(512, 3)
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# 优化器
optimizer = Optimizer_Adam(learning_rate=0.02, decay=5e-7)

# 循环10000轮
for epoch in range(10001):
    # 前向传播
```

```
        dense1.forward(X)
        activation1.forward(dense1.output)
        dense2.forward(activation1.output)
        data_loss = loss_activation.forward(dense2.output, y)
        regularization_loss = loss_activation.loss.regularization_loss(dense1)
+loss_activation.loss.regularization_loss(dense2)
        loss = data_loss + regularization_loss
        # 最高confidence的类别
        predictions = np.argmax(loss_activation.output, axis=1)
        if len(y.shape) == 2: # onehot编码
            # 改成只有一个类别
            y = np.argmax(y, axis=1)
        accuracy = np.mean(predictions == y)

        if not epoch % 100:
            print(f'epoch: {epoch}, ' +
                  f'acc: {accuracy:.3f}, ' +
                  f'loss: {loss:.3f} (' +
                  f'data_loss: {data_loss:.3f}, ' +
                  f'reg_loss: {regularization_loss:.3f}), ' +
                  f'lr: {optimizer.current_learning_rate}'
                  )

        # 反向传播
        loss_activation.backward(loss_activation.output, y)
        dense2.backward(loss_activation.dinput)
        activation1.backward(dense2.dinput)
        dense1.backward(activation1.dinput)

        # 更新梯度
        optimizer.pre_update_param()
        optimizer.update_param(dense1)
        optimizer.update_param(dense2)
        optimizer.post_update_param()



# Create test dataset

# Perform a forward pass of our testing data through this layer
dense1.forward(X_test)
# Perform a forward pass through activation function
# takes the output of first dense layer here
activation1.forward(dense1.output)
# Perform a forward pass through second Dense layer
# takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)
# Perform a forward pass through the activation/loss function
# takes the output of second dense layer here and returns loss
loss = loss_activation.forward(dense2.output, y_test)
```

```python
# Calculate accuracy from output of activation2 and targets
# calculate values along first axis
predictions = np.argmax(loss_activation.output, axis=1)
if len(y_test.shape) == 2:
      y_test = np.argmax(y_test, axis=1)
accuracy = np.mean(predictions==y_test)
print(f'validation, acc: {accuracy:.3f}, loss: {loss:.3f}')
```

```
epoch: 9600, acc: 0.918, loss: 0.253 (data_loss: 0.210, reg_loss: 0.043), lr: 0.019904468503417844
epoch: 9700, acc: 0.915, loss: 0.263 (data_loss: 0.220, reg_loss: 0.043), lr: 0.019903478083036316
epoch: 9800, acc: 0.920, loss: 0.253 (data_loss: 0.211, reg_loss: 0.043), lr: 0.019902487761213932
epoch: 9900, acc: 0.921, loss: 0.254 (data_loss: 0.211, reg_loss: 0.042), lr: 0.019901497537935988
epoch: 10000, acc: 0.916, loss: 0.262 (data_loss: 0.220, reg_loss: 0.042), lr: 0.019900507413187767
validation, acc: 0.896, loss: 0.276
```

> 这可以看到加上正则后效不好，验证集上的正确率比训练集上的还要小，说明正则化没有起到作用。还需再找一下是否代码有问题。

```
epoch: 9700, acc: 0.923, loss: 0.173 (data_loss: 0.173, reg_loss: 0.000), lr: 0.019903478083036316
epoch: 9800, acc: 0.925, loss: 0.172 (data_loss: 0.172, reg_loss: 0.000), lr: 0.019902487761213932
epoch: 9900, acc: 0.924, loss: 0.172 (data_loss: 0.172, reg_loss: 0.000), lr: 0.019901497537935988
epoch: 10000, acc: 0.924, loss: 0.175 (data_loss: 0.175, reg_loss: 0.000), lr: 0.019900507413187767
validation, acc: 0.897, loss: 0.299
```

> 上面图片是未加正则项时的结果，可以看到未加正则时的最后一轮训练准确率要比加了正则项的大，说明正则项确定可以减小训练集上的过拟合，但在测试集上表现并没有提升。

```
epoch: 10000, acc: 0.918, loss: 0.253 (data_loss: 0.210, reg_loss: 0.043),
lr: 0.019900507413187767
validation, acc: 0.920, loss: 0.256
```

> 上图是书中给出的在同样的参数设置下的训练和测试结果，可以看到lr值是一样的，说明优化器代码正确。训练集上的其他指标表现也差不多，但测试集表现却相差太大。

```
# 2输入64输出
dense1 = Layer_Dense(2, 256, weight_L2=5e-4, bias_L2=5e-4)#, weight_L2=5e-4, bias_L2=5e-4
activation1 = Activation_ReLu()
# 2输入64输出
dense2 = Layer_Dense(256, 128, weight_L2=5e-4, bias_L2=5e-4)#, weight_L2=5e-4, bias_L2=5e-4
activation2 = Activation_ReLu()
# 64输入3输出
dense3 = Layer_Dense(128, 3)
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()
```

> 增加模型复杂度，三层神经元。

```
epoch: 9500, acc: 0.924, loss: 0.240 (data_loss: 0.191, reg_loss: 0.049), lr: 0.01990545902237324
epoch: 9600, acc: 0.924, loss: 0.235 (data_loss: 0.189, reg_loss: 0.046), lr: 0.019904468503417844
epoch: 9700, acc: 0.923, loss: 0.232 (data_loss: 0.189, reg_loss: 0.043), lr: 0.019903478083036316
epoch: 9800, acc: 0.925, loss: 0.231 (data_loss: 0.188, reg_loss: 0.043), lr: 0.019902487761213932
epoch: 9900, acc: 0.926, loss: 0.228 (data_loss: 0.188, reg_loss: 0.041), lr: 0.019901497537935988
epoch: 10000, acc: 0.925, loss: 0.227 (data_loss: 0.187, reg_loss: 0.040), lr: 0.019900507413187767
validation, acc: 0.898, loss: 0.260
```

> 并没有太大提升。

```
epoch: 9500, acc: 0.934, loss: 0.156 (data_loss: 0.156, reg_loss: 0.000), lr: 0.01990545902237324
epoch: 9600, acc: 0.949, loss: 0.125 (data_loss: 0.125, reg_loss: 0.000), lr: 0.019904468503417844
epoch: 9700, acc: 0.938, loss: 0.135 (data_loss: 0.135, reg_loss: 0.000), lr: 0.019903478083036316
epoch: 9800, acc: 0.938, loss: 0.132 (data_loss: 0.132, reg_loss: 0.000), lr: 0.019902487761213932
epoch: 9900, acc: 0.944, loss: 0.125 (data_loss: 0.125, reg_loss: 0.000), lr: 0.019901497537935988
epoch: 10000, acc: 0.946, loss: 0.121 (data_loss: 0.121, reg_loss: 0.000), lr: 0.019900507413187767
validation, acc: 0.876, loss: 0.451
```

> 同样是三层结构，但不使用正则项。可以看到训练准确率更高，但测试准确率更低，说明正测项也是有效果的。但无论如何也不能像书中结果一样：测试集准确率大于训练集。