OPTIMIZER

一、内容

在这一部分，将实现Stochastic Gradient Descent（SGD）、Batch Gradient Descent（BGD）、Mini-batch Gradient Descent（MBGD）、Momentum、AdaGrad、RMSProp、Adam等优化器。

二、优化器

一、SGD

公式

$$W_{(t)} = W_{(t-1)} - \eta \cdot \nabla J$$

实现

```
class Optimizer_SGD():
      # 初始化方法将接收超参数，从学习率开始，将它们存储在类的属性中
      def __init__(self, learning_rate = 1.0):
            self.learning_rate = learning_rate

      # 给一个层对象参数，执行最基本的优化
      def update_param(self, layer):
            layer.weight += - self.learning_rate * layer.dweight
            # (64,) = (64,) + (1,64) >> (1,64)
            # (64,) += (1,64) >> 无法广播
            # (1, 64) = (64,) + (1,64) >> (1,64)
            # (1, 64) += (64,) >> (1,64)
            # 所以修改了dense中
            # self.bias = np.zeros(n_neuron) => self.bias = np.zeros((1, n_neuron))
            layer.bias += - self.learning_rate * layer.dbias
```

实例

```
# 数据集
X, y = spiral_data(samples=100, classes=3)

# 2输入64输出
dense1 = Layer_Dense(2, 64)
activation1 = Activation_ReLu()
# 64输入3输出
dense2 = Layer_Dense(64, 3)
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# 优化器
optimizer = Optimizer_SGD()

# 循环10000轮
for epoch in range(10001):
    # 前向传播
    dense1.forward(X)
    activation1.forward(dense1.output)
    dense2.forward(activation1.output)
    loss = loss_activation.forward(dense2.output, y)

    # 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}')

    # 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.update_param(dense1)
    optimizer.update_param(dense2)
```

```
epoch: 0, acc: 0.363, loss: 1.099
epoch: 100, acc: 0.440, loss: 1.075
epoch: 200, acc: 0.440, loss: 1.066
epoch: 300, acc: 0.457, loss: 1.065
epoch: 400, acc: 0.450, loss: 1.064
```

```
epoch: 9600, acc: 0.687, loss: 0.654
epoch: 9700, acc: 0.683, loss: 0.713
epoch: 9800, acc: 0.710, loss: 0.625
epoch: 9900, acc: 0.673, loss: 0.646
epoch: 10000, acc: 0.683, loss: 0.661
```

> 可以看到准确率提高了，损失下降了。

公式

学习率衰减的目的是在训练过程中逐渐减小学习率。这样做的原因是，使用一个固定的学习率来训练神经网络，并且最终会在远离实际最小值的地方振荡。为了克服这种情况，在训练过程中逐渐减小学习率的建议，这有助于网络收敛到局部最小值并避免振荡。每一步更新学习率，取步数分数的倒数。称为学习率衰减。这种衰减的工作原理是取步数和衰减比率并将它们相乘。训练越深入，步数越大，这个乘法的结果也越大。然后我们取它的倒数（训练越深入，值越低），并将初始学习率乘以它。添加的 1 确保结果算法永远不会提高学习率。

$$r_c = \frac{r}{(1 + decay \times t)}$$

> t是epoch数量

实现

```python
class Optimizer_SGD():
    # 初始化方法将接收超参数，从学习率开始，将它们存储在类的属性中
    def __init__(self, learning_rate = 1.0, decay = 0):
        self.learning_rate = learning_rate
        self.decay = decay
        self.current_learning_rate = learning_rate
        self.iteration = 0

    def pre_update_param(self):
```

```
            # 这种衰减的工作原理是取步数和衰减比率并将它们相乘。
            if self.decay:
                  self.current_learning_rate = self.learning_rate * \
                                                      (1 / (1 + self.decay *
self.iteration))

      # 给一个层对象参数，执行最基本的优化
      def update_param(self, layer):
            layer.weight += - self.current_learning_rate * layer.dweight
            # (64,) = (64,) + (1,64) >> (1,64)
            # (64,) += (1,64) >> 无法广播
            # (1, 64) = (64,) + (1,64) >> (1,64)
            # (1, 64) += (64,) >> (1,64)
            # 所以修改了dense中
            # self.bias = np.zeros(n_neuron) => self.bias = np.zeros((1, n_neuron))
            layer.bias += - self.current_learning_rate * layer.dbias

      def post_update_param(self):
            self.iteration += 1
```

实例

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

```
epoch: 9500, acc: 0.697, loss: 0.708, lr: 0.09524716639679968
epoch: 9600, acc: 0.700, loss: 0.704, lr: 0.09434852344560807
epoch: 9700, acc: 0.700, loss: 0.701, lr: 0.09346667912889055
epoch: 9800, acc: 0.697, loss: 0.698, lr: 0.09260116677470137
epoch: 9900, acc: 0.697, loss: 0.696, lr: 0.09175153683824203
epoch: 10000, acc: 0.693, loss: 0.693, lr: 0.09091735612328393
```

二、Momentum

公式

$$如果令V_{(t)} = \beta \cdot V_{(t-1)} + (1-\beta) \cdot \Delta W_{(t)i}$$

$$W_{(t)i} = W_{(t-1)i} - \eta \cdot V_{(t)}$$

在实现的时候，$(1-\beta)$直接取为1

实现

这里对Momentum优化器的实现并不是重新实现一个优化器，而是在SGD的基础上，通过momentum参数调用。

```
class Optimizer_SGD():
      # 初始化方法将接收超参数，从学习率开始，将它们存储在类的属性中
      def __init__(self, learning_rate = 1.0, decay = 0, momentum=0):
            self.learning_rate = learning_rate
            self.decay = decay
            self.current_learning_rate = learning_rate
            self.iteration = 0
            self.momentum = momentum

      def pre_update_param(self):
            # 这种衰减的工作原理是取步数和衰减比率并将它们相乘。
            if self.decay:
                  self.current_learning_rate = self.learning_rate * \
                                                (1 / (1 + self.decay *
self.iteration))

      # 给一个层对象参数，执行最基本的优化
      def update_param(self, layer):

            deta_weight = layer.dweight
            deta_bias = layer.dbias

            # 如果使用momentum
            if self.momentum:
                  # 如果还没有累积动量
                  if not hasattr(layer, "dweight_cumulate"):
                        # 注意：这里是往layer层里加属性
                        # 这很容易理解，历史信息肯定是要存在对应的对像中
                        layer.dweight_cumulate = np.zeros_like(layer.weight)
```

```
                layer.dbias_cumulate = np.zeros_like(layer.bias)
            deta_weight += self.momentum * layer.dweight_cumulate
            layer.dweight_cumulate = deta_weight
            deta_bias += self.momentum * layer.dbias_cumulate
            layer.dbias_cumulate = deta_bias
        layer.weight -= self.current_learning_rate * deta_weight
        # (64,) = (64,) + (1,64) >> (1,64)
        # (64,) += (1,64) >> 无法广播
        # (1, 64) = (64,) + (1,64) >> (1,64)
        # (1, 64) += (64,) >> (1,64)
        # 所以修改了dense中
        # self.bias = np.zeros(n_neuron) => self.bias = np.zeros((1, n_neuron))
        layer.bias -= self.current_learning_rate * deta_bias

    def post_update_param(self):
        self.iteration += 1
```

实例

```
# 数据集
X, y = spiral_data(samples=100, classes=3)

# 2输入64输出
dense1 = Layer_Dense(2, 64)
activation1 = Activation_ReLu()
# 64输入3输出
dense2 = Layer_Dense(64, 3)
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# 优化器
optimizer = Optimizer_SGD(decay=0.001, momentum=0.8)

# 循环10000轮
for epoch in range(10001):
    # 前向传播
    dense1.forward(X)
    activation1.forward(dense1.output)
    dense2.forward(activation1.output)
    loss = loss_activation.forward(dense2.output, y)

    # 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'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()
```

这里取momentum=0.8

```
epoch: 9600, acc: 0.950, loss: 0.165, lr: 0.09434852344560807
epoch: 9700, acc: 0.950, loss: 0.164, lr: 0.09346667912889055
epoch: 9800, acc: 0.950, loss: 0.164, lr: 0.09260116677470137
epoch: 9900, acc: 0.950, loss: 0.163, lr: 0.09175153683824203
epoch: 10000, acc: 0.950, loss: 0.162, lr: 0.09091735612328393
```

可以看到准确率提高到了95%,loss低到了0.16

三、Adagrad

AdaGrad，即自适应梯度，是一种为每个参数设定学习率而不是全局共享率的方法，**为每个参数计算一个自适应的学习率。**这里的想法是对特征进行归一化更新。在训练过程中，有些权重可能会显著增加，而有些权重则不会改变太多。由于更新的单调性，用一个不断增加的缓存进行除法运算也可能导致学习停滞，因为随着时间的推移，更新变得越来越小。这就是为什么这个优化器除了一些特定的应用之外，没有被广泛使用的原因。这个优化器通常用在稀疏数据上（特征特别多），主要是特征不同，而不是特征的程度不同的数据上。

[“随机梯度下降、牛顿法、动量法、Nesterov、AdaGrad、RMSprop、Adam”，打包理解对梯度下降法的优化*哔哩哔哩*bilibili](#))

公式

$$W_{(t)i} = W_{(t-1)i} - \frac{\eta}{\sqrt{S_{(t)}} + \varepsilon} \cdot \Delta W_{(t)i}$$

$$\text{其中：} S_{(t)} = S_{(t-1)} + \Delta W_{(t)i} \cdot \Delta W_{(t)i}$$

$$\Delta W_{(t)i} = \frac{\partial J(W_{(t-1)i})}{\partial W_i}$$

实现

```
class Optimizer_Adagrad():
      # 初始化方法将接收超参数，从学习率开始，将它们存储在类的属性中
      def __init__(self, learning_rate = 1.0, decay = 0, epsilon = 1e-7):
            self.learning_rate = learning_rate
            self.decay = decay
            self.current_learning_rate = learning_rate
            self.iteration = 0
            # 极小值，防止除以0
            self.epsilon = epsilon


      def pre_update_param(self):
            # 这种衰减的工作原理是取步数和衰减比率并将它们相乘。
            if self.decay:
                  self.current_learning_rate = self.learning_rate * \
                                                (1 / (1 + self.decay *
self.iteration))

      # 给一个层对象参数
      def update_param(self, layer):
            if not hasattr(layer, 'dweight_square_sum'):
                  layer.dweight_square_sum = np.zeros_like(layer.weight)
```

```
                layer.dbias_square_sum = np.zeros_like(layer.bias)
            layer.dweight_square_sum += layer.dweight ** 2
            layer.dbias_square_sum += layer.dbias ** 2
            layer.weight += -self.current_learning_rate * layer.dweight / \
                            ( np.sqrt(layer.dweight_square_sum) + self.epsilon )
            layer.bias += -self.current_learning_rate * layer.dbias / \
                            (np.sqrt(layer.dbias_square_sum) + self.epsilon)

    def post_update_param(self):
            self.iteration += 1
```

实例

```
epoch: 9600, acc: 0.743, loss: 0.526, lr: 0.09434852344560807
epoch: 9700, acc: 0.740, loss: 0.526, lr: 0.09346667912889055
epoch: 9800, acc: 0.740, loss: 0.525, lr: 0.09260116677470137
epoch: 9900, acc: 0.740, loss: 0.524, lr: 0.09175153683824203
epoch: 10000, acc: 0.747, loss: 0.523, lr: 0.09091735612328393
```

AdaGrad在这里表现得相当不错，但没有SGD with momentum好，我们可以看到损失在整个训练过程中一直在下降。有趣的是，AdaGrad最初花了更多的周期才达到和带有动量的随机梯度下降相似的结果。这可能是因为AdaGrad的学习率随着梯度的累积而逐渐减小，导致后期的更新变得很小，而SGD with momentum则能够保持一定的更新速度和方向。不过，AdaGrad也有它的优势，比如能够处理稀疏数据和不同尺度的特征。

四、RMSProp

RMSProp（Root Mean Square Propagation）和AdaGrad类似，RMSProp也是**为每个参数计算一个自适应的学习率**；它只是用一种不同于AdaGrad的方式来计算。RMSProp的主要思想是使用一个指数衰减的平均来存储过去梯度的平方，从而避免了AdaGrad学习率过快下降的问题。

公式

$$S_{(t)} = \beta S_{(t-1)} + (1-\beta)\Delta W_{(t)i} \cdot \Delta W_{(t)i}$$

$$W_{(t)i} = W_{(t-1)i} - \frac{\eta}{\sqrt{S_{(t)}} + \varepsilon} \cdot \Delta W_{(t)i}$$

RMSProp与Adagrad加入了一个加权系数，这里的新超参数是β。β是缓存记忆衰减率，越早的梯度平方占的权得越低。由于这个优化器在默认值下，能够保持很大的自适应学习率更新，所以即使很小的梯度更新也足以让它继续运行（梯度减小的慢）；因此，默认学习率为1太大了，会导致模型立刻不稳定。一个能够再次稳定并且给出足够快速更新的学习率大约是0.001（这也是一些知名机器学习框架中使用的这个优化器的默认值）。我们从现在开始也会用这个值作为默认值。

实现

```
class Optimizer_RMSprop():
      # 初始化方法将接收超参数，从学习率开始，将它们存储在类的属性中
      def __init__(self, learning_rate = 0.001, decay = 0, epsilon = 1e-7, beta = 0.9):
            # 注意：这里的学习率learning_rate = 0.001，不是默认为1
            self.learning_rate = learning_rate
            self.decay = decay
            self.current_learning_rate = learning_rate
            self.iteration = 0
            # 极小值，防止除以0
            self.epsilon = epsilon
            self.beta = beta

      def pre_update_param(self):
            # 这种衰减的工作原理是取步数和衰减比率并将它们相乘。
            if self.decay:
                  self.current_learning_rate = self.learning_rate * \
                                                (1 / (1 + self.decay *
self.iteration))

      # 给一个层对象参数
      def update_param(self, layer):
            if not hasattr(layer, 'dweight_square_sum'):
                  layer.dweight_square_sum = np.zeros_like(layer.weight)
                  layer.dbias_square_sum = np.zeros_like(layer.bias)
```

```
            layer.dweight_square_sum = self.beta * layer.dweight_square_sum + (1 -
self.beta) * layer.dweight ** 2
            layer.dbias_square_sum = self.beta * layer.dbias_square_sum + (1 - self.beta)
* layer.dbias ** 2
            layer.weight += -self.current_learning_rate * layer.dweight / \
                                    ( np.sqrt(layer.dweight_square_sum) + self.epsilon )
            layer.bias += -self.current_learning_rate * layer.dbias / \
                                    (np.sqrt(layer.dbias_square_sum) + self.epsilon)

    def post_update_param(self):
            self.iteration += 1
```

实例

```
epoch: 8100, acc: 0.663, loss: 0.810, lr: 0.001
epoch: 8200, acc: 0.670, loss: 0.808, lr: 0.001
epoch: 8300, acc: 0.667, loss: 0.807, lr: 0.001
epoch: 8400, acc: 0.670, loss: 0.805, lr: 0.001
epoch: 8500, acc: 0.667, loss: 0.803, lr: 0.001
epoch: 8600, acc: 0.663, loss: 0.801, lr: 0.001
epoch: 8700, acc: 0.663, loss: 0.798, lr: 0.001
epoch: 8800, acc: 0.667, loss: 0.796, lr: 0.001
epoch: 8900, acc: 0.670, loss: 0.792, lr: 0.001
epoch: 9000, acc: 0.667, loss: 0.787, lr: 0.001
epoch: 9100, acc: 0.657, loss: 0.784, lr: 0.001
epoch: 9200, acc: 0.663, loss: 0.781, lr: 0.001
epoch: 9300, acc: 0.660, loss: 0.778, lr: 0.001
epoch: 9400, acc: 0.667, loss: 0.775, lr: 0.001
epoch: 9500, acc: 0.657, loss: 0.771, lr: 0.001
epoch: 9600, acc: 0.663, loss: 0.766, lr: 0.001
epoch: 9700, acc: 0.670, loss: 0.762, lr: 0.001
epoch: 9800, acc: 0.663, loss: 0.758, lr: 0.001
epoch: 9900, acc: 0.660, loss: 0.756, lr: 0.001
epoch: 10000, acc: 0.657, loss: 0.753, lr: 0.001
```

可以看到学习率和loss变化很慢。

```
# 优化器
optimizer = Optimizer_RMSprop(learning_rate=0.02, decay=1e-5,beta=0.999)
```

```
epoch: 9600, acc: 0.880, loss: 0.251, lr: 0.0182483416819496 54
epoch: 9700, acc: 0.873, loss: 0.257, lr: 0.0182317067612284 56
epoch: 9800, acc: 0.873, loss: 0.259, lr: 0.0182151021411852 55
epoch: 9900, acc: 0.880, loss: 0.260, lr: 0.0181985277391059 07
epoch: 10000, acc: 0.880, loss: 0.260, lr: 0.0181819834725770 25
```

这个优化器参数不好调，因为改变了β，$(1-\beta)$也改变了，两者都有影响。

五、Adam

Adam（Adaptive Momentum），即自适应动量，目前是最广泛使用的优化器，它建立在RMSProp之上，并加入了SGD中的动量概念。这意味着，我们不再直接应用当前的梯度，而是像带有动量的SGD优化器一样应用动量，然后像RMSProp一样用缓存来应用每个权重的自适应学习率。这样，我们就能够结合SGD和RMSProp的优点，实现更快、更稳定的训练过程。

公式

$$V_{(t)} = \beta_1 \cdot V_{(t-1)} + (1-\beta_1) \cdot \Delta W_{(t)i}$$

$$S_{(t)} = \beta_2 S_{(t-1)} + (1-\beta_2) \Delta W_{(t)i} \cdot \Delta W_{(t)i}$$

$$W_{(t)i} = W_{(t-1)i} - \frac{\eta}{\sqrt{S_{(t)}} + \varepsilon} \cdot V_{(t)}$$

在训练开始时，动量和缓存的初始值通常都是0，这会导致训练速度较慢。为了解决这个问题，可以使用偏差校正机制来对动量还有平方和进行修正。偏差校正机制的原理是将动量、平方和除以一个衰减系数，这个系数随着训练的进行而逐渐减小，最终趋近于1。在训练初期，由于衰减系数较大，所以除以它会使动量和缓存变得更大，从而加快训练速度。随着训练的进行，衰减系数逐渐减小，动量和缓存也会逐渐恢复到正常值。

$$1 - \beta^t$$

> 这就是衰减系数，t是epoch数，开始时衰减系数很小，除以它能得到一个很大的数，所以在开始时梯度下降很快。

实现

```
# 数据集
X, y = spiral_data(samples=100, classes=3)

# 2输入64输出
dense1 = Layer_Dense(2, 64)
activation1 = Activation_ReLu()
# 64输入3输出
dense2 = Layer_Dense(64, 3)
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# 优化器
optimizer = Optimizer_Adam(learning_rate=0.05, 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)
    loss = loss_activation.forward(dense2.output, y)

    # 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'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()
```

```
epoch: 9500, acc: 0.807, loss: 0.402, lr: 0.0497636475559331
epoch: 9600, acc: 0.813, loss: 0.405, lr: 0.0497611711258544616
epoch: 9700, acc: 0.817, loss: 0.404, lr: 0.0497586952075908
epoch: 9800, acc: 0.823, loss: 0.400, lr: 0.04975621940303483
epoch: 9900, acc: 0.820, loss: 0.395, lr: 0.04975537438448399965
epoch: 10000, acc: 0.817, loss: 0.399, lr: 0.04975126853296942
```

虽然Adam在这里表现得最好，通常也是最好的优化器之一，但并不总是这样。通常先尝试Adam优化器是个好主意，但也要尝试其他优化器，特别是当你没有得到期望的结果时。有时简单的SGD或SGD + 动量比Adam表现得更好。原因各不相同，但请记住这一点。 我们将在训练时介绍如何选择各种超参数（如学习率），但对于SGD来说，一个通常的初始学习率是1.0，衰减到0.1。对于Adam来说，一个好的初始LR是0.001（1e-3），衰减到0.0001（1e-4）。不同的问题可能需要不同的值，但这些值都是一个不错的起点。