DROPOUT

一、内容

神经网络正则化的另一种选择是添加一个dropout层，它禁用一些神经元，而其他神经元保持不变。这里的想法与正则化类似，是为了防止神经网络过于依赖任何神经元或在特定实例中完全依赖任何神经元。Dropout函数通过在每次前向传递期间以给定速率随机禁用神经元来工作，迫使网络学习如何仅使用剩余的随机部分神经元进行准确预测。Dropout迫使模型为同一目的使用更多的神经元，从而增加了学习描述数据的底层函数的机会。例如，如果在当前步骤中禁用一半的神经元，在下一步中禁用另一半，则强迫更多的神经元学习数据，因为只有它们中的一部分“看到”数据并在给定传递中获得更新。这些交替的神经元半数只是一个例子，将使用一个超参数来通知dropout层随机禁用多少个神经元。**dropout层并不真正禁用神经元，而是将它们的输出归零。换句话说，dropout并不减少使用的神经元数量，也不会在禁用一半神经元时使训练过程快两倍。**

二、代码

函数

代码将使用np.random.binomial()函数实现对dropout概率的设定。

```
np.random.binomial(2, 0.8, size=10)
```

np.random.binomial是NumPy库中的一个函数，它用于从二项分布中抽取样本。二项分布是一种离散概率分布，它描述了在 n 次独立的是/非试验中成功的次数，其中每次试验的成功概率为p。函数接受三个参数：n、p和size。n表示试验次数，p表示每次试验的成功概率，size表示要抽取的样本数量。例如，上面的代码将从一个参数为n=2和p=0.8的二项分布中抽取10个样本。

公式

$$Dr_i = \begin{cases} \frac{z_i}{1-q} & r_i = 1 \\ 0 & r_i = 0 \end{cases} \to \frac{\partial}{\partial z_i} Dr_i = \begin{cases} \frac{1}{1-q} & r_i = 1 \\ 0 & r_i = 0 \end{cases}$$

> Dr是dropout函数，z是输入，q是断开连接的概率。因为0的概率只有q这么大，就要除以$1-q$偿损失值。

$$\frac{\partial loss}{\partial z} = \frac{\partial loss}{\partial Dr} \frac{\partial Dr}{\partial z}$$

实现

```
class Dropout():
    def __init__(self, rate):
        # rate是断开连接的概率
        self.rate = 1 - rate

    def forward(self, input):
        self.input = input
        # 按概率生成一个0、1矩阵
        # 因为1的概率只有rate这么大，就要除以rate偿损失值
        self.mask = np.random.binomial(1, self.rate, size=self.input.shape) / 
self.rate
        self.output = self.input * self.mask

    def backward(self,dvalue):
        self.dinput = dvalue * self.mask
```

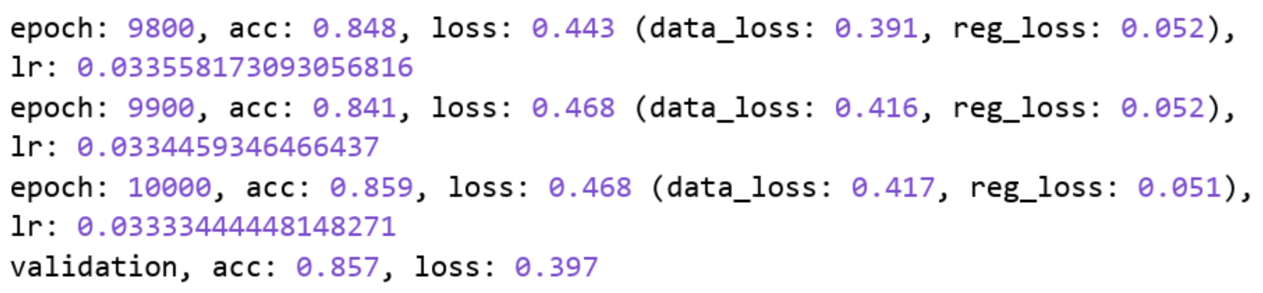
实例

```
# 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)
activation1 = Activation_ReLu()

dropout1 = Dropout(0.1)

# 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=0.05, decay=5e-5)

# 循环10000轮
for epoch in range(10001):
      # 前向传播
      dense1.forward(X)
      activation1.forward(dense1.output)
      dropout1.forward(activation1.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)
    dropout1.backward(dense2.dinput)
    activation1.backward(dropout1.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)
# 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: 9500, acc: 0.914, loss: 0.282 (data_loss: 0.224, reg_loss: 0.058), lr: 0.03389945421878708
epoch: 9600, acc: 0.916, loss: 0.281 (data_loss: 0.224, reg_loss: 0.057), lr: 0.03378492516639756
epoch: 9700, acc: 0.914, loss: 0.279 (data_loss: 0.223, reg_loss: 0.056), lr: 0.03367116737937304
epoch: 9800, acc: 0.912, loss: 0.288 (data_loss: 0.227, reg_loss: 0.062), lr: 0.033558173093056816
epoch: 9900, acc: 0.916, loss: 0.279 (data_loss: 0.222, reg_loss: 0.057), lr: 0.0334459346466437
epoch: 10000, acc: 0.914, loss: 0.277 (data_loss: 0.222, reg_loss: 0.055), lr: 0.03333444448148271
validation, acc: 0.897, loss: 0.299
```



```
epoch: 9800, acc: 0.848, loss: 0.443 (data_loss: 0.391, reg_loss: 0.052),
lr: 0.033558173093056816
epoch: 9900, acc: 0.841, loss: 0.468 (data_loss: 0.416, reg_loss: 0.052),
lr: 0.0334459346466437
epoch: 10000, acc: 0.859, loss: 0.468 (data_loss: 0.417, reg_loss: 0.051),
lr: 0.03333444448148271
validation, acc: 0.857, loss: 0.397
```

> 实际结果要比书中的好。