

REGRESSION

一、内容

本部分将实现能解决回归问题的模型。

二、代码

一、Linear Activation

这个线性激活函数不修改它的输入，而是将它传递到输出： $y = x$ 。对于反向传递，我们已经知道 $f(x) = x$ 的导数是1。做只是为了完整性和清晰性，以便在模型定义代码中看到输出层的激活函数。从计算时间的角度来看，这几乎不会增加处理时间，至少不足以明显影响训练时间。

实现

```
class Activation_Linear:
    def __init__(self):
        pass

    def forward(self, input):
        self.input = input
        self.output = self.input

    def backward(self, dvalue):
        # 注意不能self.dinput = dvalue
        # 这意味着 dinput 和 dvalue 指向同一个对象，因此对 dinput 的任何更改都会影响原始的 dvalue 对象
        # 而对dvalue进行运算如乘1，则和下面代码一样
        self.dinput = dvalue.copy()
```

二、Mean Squared Error Loss

公式

$$L_i = \frac{1}{J} \sum_j (y_{i,j} - \hat{y}_{i,j})^2$$

$$\begin{aligned} \frac{\partial}{\partial \hat{y}_{i,j}} L_i &= \frac{\partial}{\partial \hat{y}_{i,j}} \left[\frac{1}{J} \sum_j (y_{i,j} - \hat{y}_{i,j})^2 \right] = \frac{1}{J} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} (y_{i,j} - \hat{y}_{i,j})^2 = \\ &= \frac{1}{J} \cdot 2 \cdot (y_{i,j} - \hat{y}_{i,j})^{2-1} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} [y_{i,j} - \hat{y}_{i,j}] = \\ &= \frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j})^1 \cdot \left(\frac{\partial}{\partial \hat{y}_{i,j}} y_{i,j} - \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j} \right) = \frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j}) \cdot (0 - 1) = \\ &= \frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j}) \cdot (-1) = -\frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j}) \end{aligned}$$

公式都很好理解，不做过多解释。

三、回归问题中衡量准确率

在交叉熵中，可以计算匹配的数量（预测等于真实目标的情况），然后除以样本数来衡量模型的准确度。在回归模型中，预测是一个浮点值，不能简单地检查输出值是否等于真实值，因为它很可能不会——如果它稍微不同，准确度就会是0。对于回归来说，没有完美的方法来显示准确度。不过，最好还是有一些准确度指标。例如，Keras，一个流行的深度学习框架，会显示回归模型的准确度和损失，我们也会制作自己的准确度指标。**计算真实目标值的标准差，然后除以250。这个值可以根据目标而变化。除以的数字越大，准确度指标就越“严格”。250是这里选择的值。**

```
accuracy_precision = np.std(y) / 250
predictions = activation2.output
accuracy = np.mean(np.absolute(predictions - y) < accuracy_precision)
```

实例

```
# 生成数据共1000个点
X, y = sine_data()
X_test = X[500:]
y_test = y[500:]
X = X[0:500]
y = y[0:500]

# 三层结构
dense1 = Layer_Dense(1, 64)
activation1 = Activation_ReLu()
dense2 = Layer_Dense(64, 64) # , weight_L2=1e-4, bias_L2=1e-4
activation2 = Activation_ReLu()
dense3 = Layer_Dense(64, 1)
activation3 = Activation_Linear()
loss_function = Loss_MeanSquaredError()

# 优化器
optimizer = Optimizer_Adam(learning_rate=0.01, decay=1e-3)

# 精度标准
accuracy_precision = np.std(y) / 250

for epoch in range(10001):
    # 前向传播
    dense1.forward(X)
    activation1.forward(dense1.output)
    dense2.forward(activation1.output)
    activation2.forward(dense2.output)
    dense3.forward(activation2.output)
    activation3.forward(dense3.output)
    data_loss = loss_function.calculate(activation3.output, y)

    regularization_loss = \
        loss_function.regularization_loss(dense1) + \
        loss_function.regularization_loss(dense2) + \
        loss_function.regularization_loss(dense3)

    loss = data_loss + regularization_loss

    # 计算准确率
    predictions = activation3.output
    accuracy = np.mean(np.absolute(predictions - y) <
                        accuracy_precision)

    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_function.backward(activation3.output, y)
activation3.backward(loss_function.dinput)
dense3.backward(activation3.dinput)
activation2.backward(dense3.dinput)
dense2.backward(activation2.dinput)
activation1.backward(dense2.dinput)
dense1.backward(activation1.dinput)

# 更新权重
optimizer.pre_update_param()
optimizer.update_param(dense1)
optimizer.update_param(dense2)
optimizer.update_param(dense3)
optimizer.post_update_param()

# 测试集
X_test, y_test = sine_data()

dense1.forward(X_test)
activation1.forward(dense1.output)
dense2.forward(activation1.output)
activation2.forward(dense2.output)
dense3.forward(activation2.output)
activation3.forward(dense3.output)

plt.plot(X_test, y_test)
plt.plot(X_test, activation3.output)
plt.show()

```

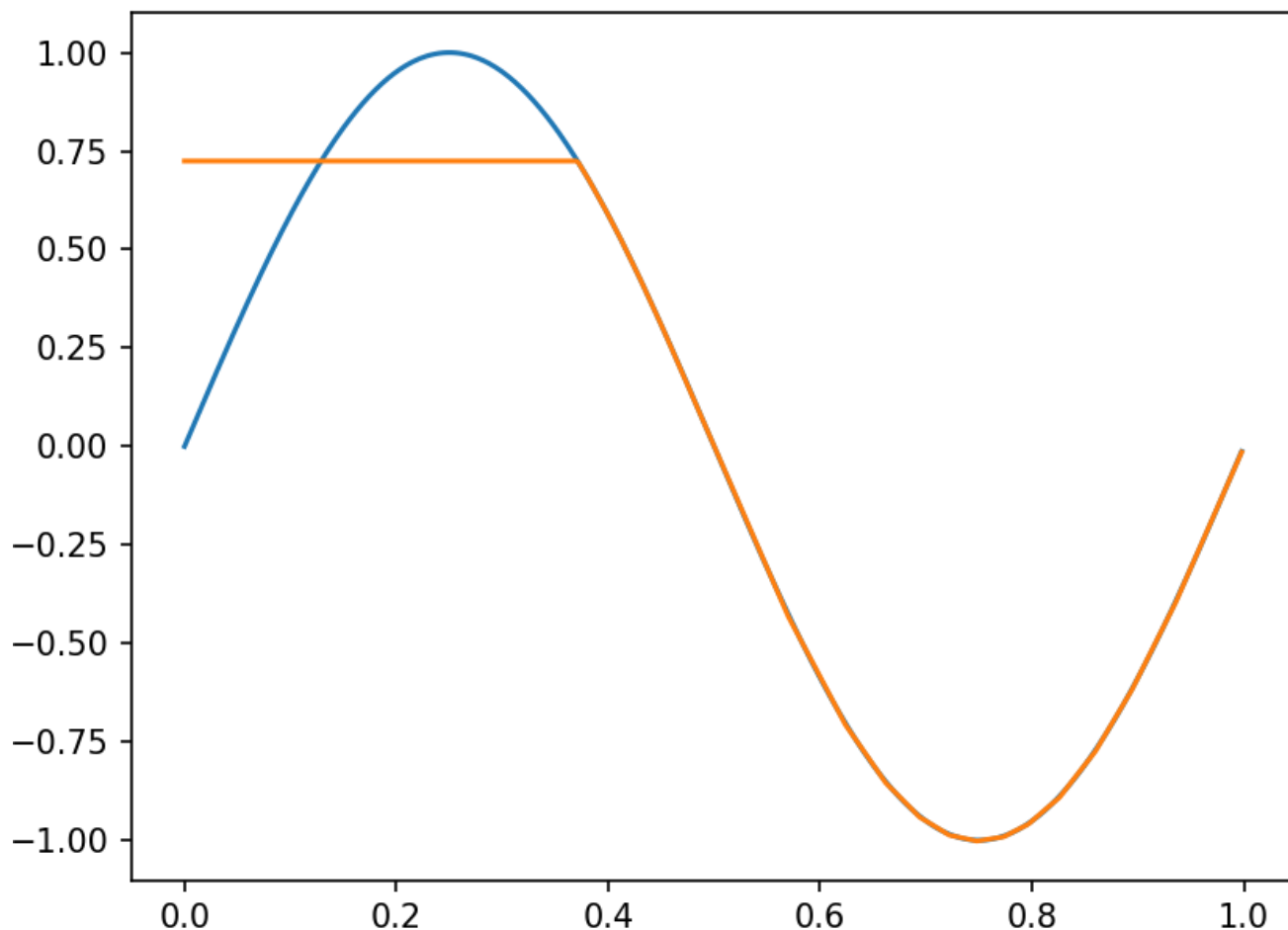
参数1

```
optimizer = Optimizer_Adam(learning_rate=0.01, decay=1e-3)
```

```

epoch: 9500, acc: 0.606, loss: 0.030 (data_loss: 0.030, reg_loss: 0.000), lr: 0.0009524716639679969
epoch: 9600, acc: 0.560, loss: 0.030 (data_loss: 0.030, reg_loss: 0.000), lr: 0.0009434852344560807
epoch: 9700, acc: 0.606, loss: 0.030 (data_loss: 0.030, reg_loss: 0.000), lr: 0.0009346667912889055
epoch: 9800, acc: 0.604, loss: 0.030 (data_loss: 0.030, reg_loss: 0.000), lr: 0.0009260116677470137
epoch: 9900, acc: 0.264, loss: 0.030 (data_loss: 0.030, reg_loss: 0.000), lr: 0.0009175153683824203
epoch: 10000, acc: 0.604, loss: 0.030 (data_loss: 0.030, reg_loss: 0.000), lr: 0.0009091735612328393

```

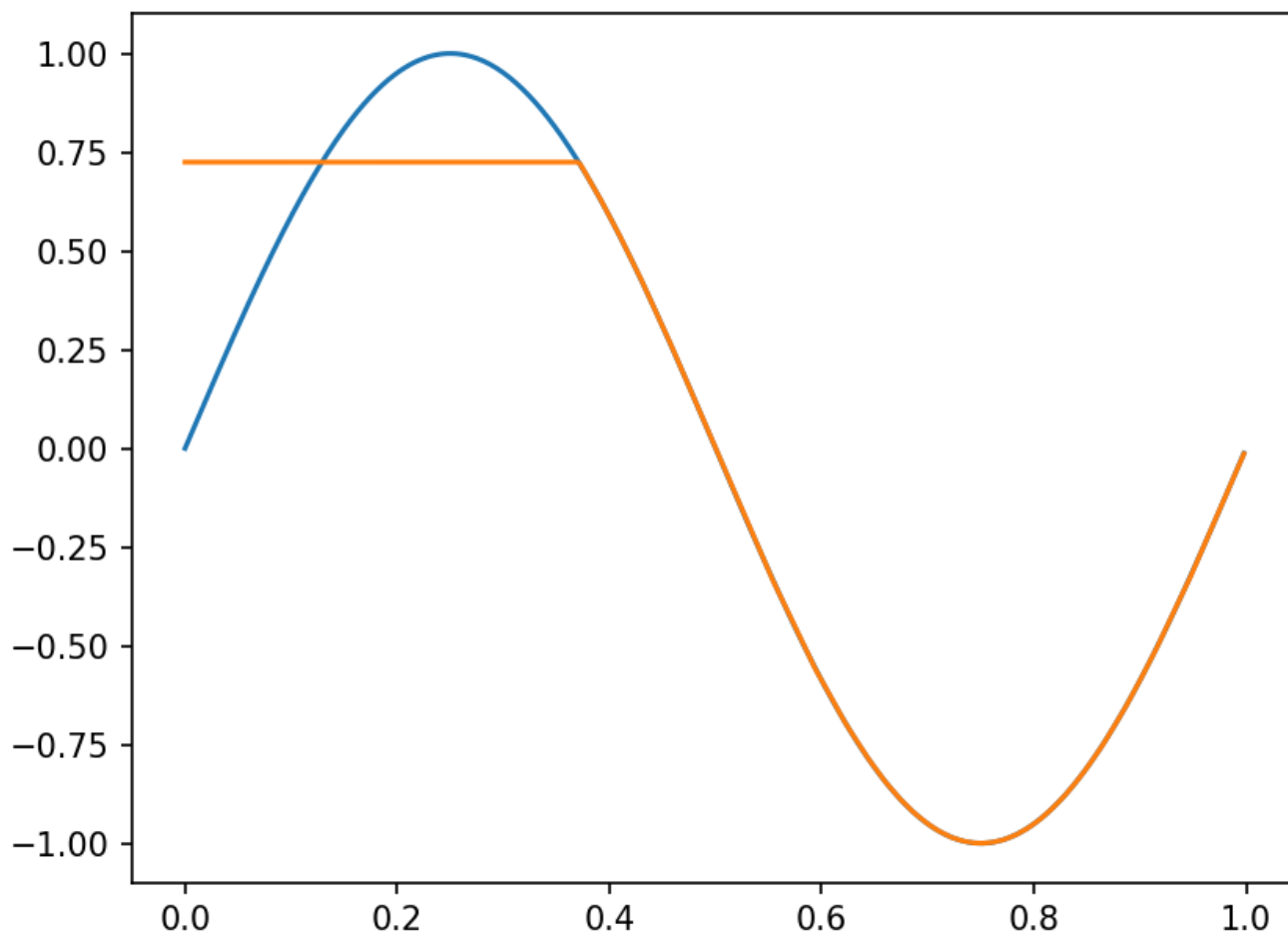


橙色线是预测值，蓝色线是ground truth，结果和书上一致。

参数2

```
optimizer = Optimizer_Adam(learning_rate = 0.001)
```

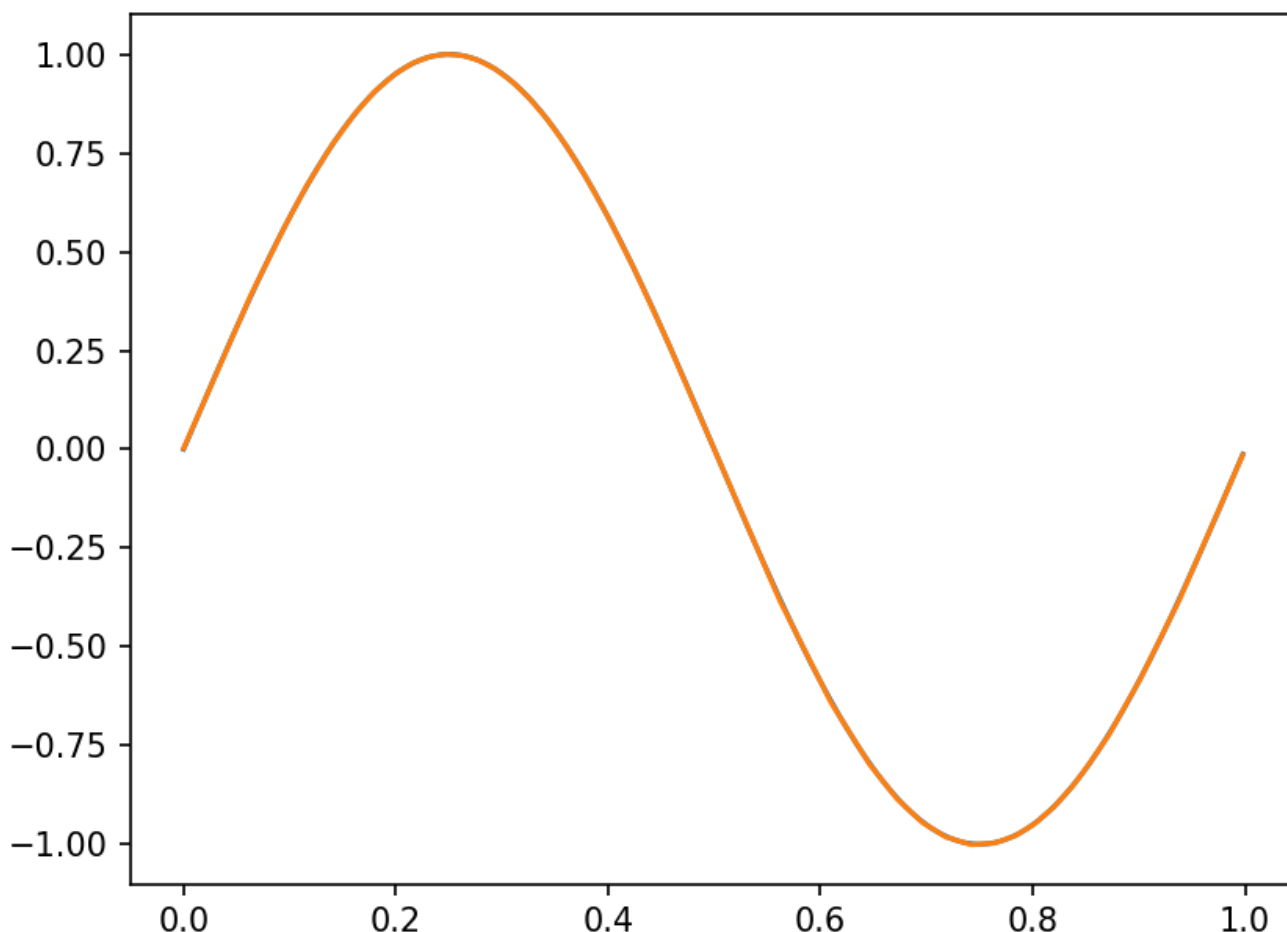
```
epoch: 9500, acc: 0.632, loss: 0.030 (data_loss: 0.030, reg_loss: 0.000), lr: 0.001
epoch: 9600, acc: 0.632, loss: 0.030 (data_loss: 0.030, reg_loss: 0.000), lr: 0.001
epoch: 9700, acc: 0.232, loss: 0.030 (data_loss: 0.030, reg_loss: 0.000), lr: 0.001
epoch: 9800, acc: 0.632, loss: 0.030 (data_loss: 0.030, reg_loss: 0.000), lr: 0.001
epoch: 9900, acc: 0.630, loss: 0.030 (data_loss: 0.030, reg_loss: 0.000), lr: 0.001
epoch: 10000, acc: 0.628, loss: 0.030 (data_loss: 0.030, reg_loss: 0.000), lr: 0.001
```



也与书上结果一致。

```
optimizer = Optimizer_Adam(learning_rate=0.005, decay=1e-3)
```

```
epoch: 9600, acc: 0.978, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr: 0.00047174261722804036
epoch: 9700, acc: 0.986, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr: 0.00046733339564445275
epoch: 9800, acc: 0.298, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr: 0.00046300583387350687
epoch: 9900, acc: 0.990, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr: 0.00045875768419121016
epoch: 10000, acc: 0.992, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr: 0.00045458678061641964
```



结果也与书上一致，`learning_rate=0.005`是最优的参数，0.01和0.001效果都不好，在此之间取0.005有效果。可以看到，在很窄的参数取值才有效果。

```
# 用正态分布初始化权重
self.weight = 0.01 * np.random.randn(n_input, n_neuron)
```

上面代码是Layer_Dense中初始化权重的代码。

可以参考Keras中的初始化实现。其中用了Glorot均匀初始化器，也称为Xavier均匀初始化器。它从一个均匀分布中抽取样本，范围在 $[-limit, limit]$ 之间，其中 $limit$

是 $\sqrt{6/(n_{input} + n_{output})}$ ， n_{input} 是权重张量中输入单元的数量，

n_{output} 是权重张量中输出单元的数量。简单来说，这种初始化方法可以根据权重张量的输入和输出单元数量来确定初始化范围，从而更好地初始化神经网络的权重。

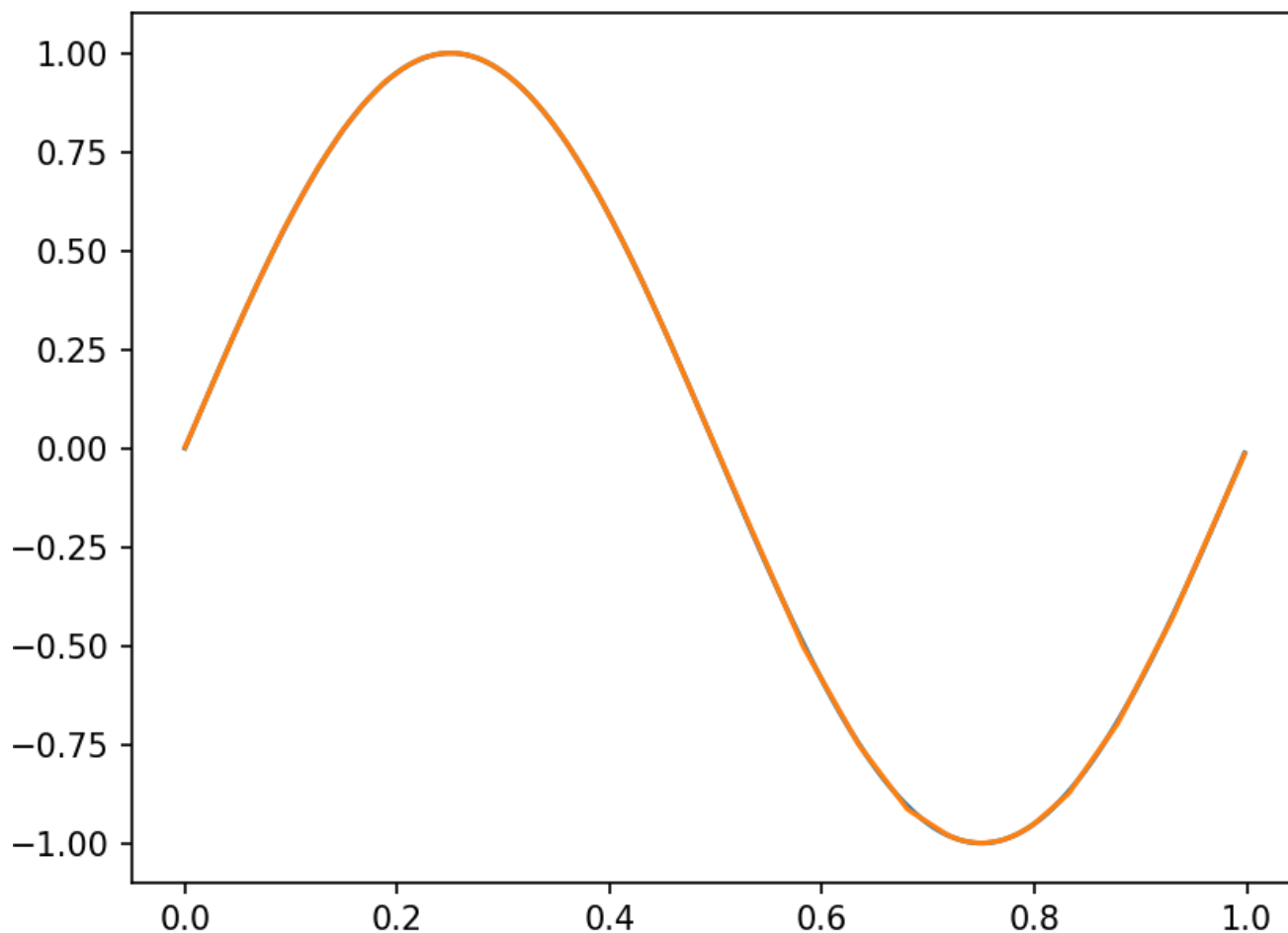
实际上在此时遇到了一个非常类似的问题，改变权重的初始化方式使模型从完全不学习到学习状态，但不按Glorot均匀初始化，只是在Layer_Dense中初始化权重的代码修改一下。为了这个目的，将Dense层的权重初始化中乘以正态分布抽取的因子改为0.1。

参数3

```
self.weight = 0.1 * np.random.randn(n_input, n_neuron)
```

```
optimizer = Optimizer_Adam(learning_rate = 0.001)
```

```
epoch: 9500, acc: 0.896, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr: 0.001  
epoch: 9600, acc: 0.792, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr: 0.001  
epoch: 9700, acc: 0.896, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr: 0.001  
epoch: 9800, acc: 0.922, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr: 0.001  
epoch: 9900, acc: 0.898, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr: 0.001  
epoch: 10000, acc: 0.922, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr: 0.001
```



将Dense层的权重初始化中乘以正态分布抽取的因子改为0.1后，用之前同样的参数训练效果还是很好。