计算机科学与技术学院神经网络与深度学习课程实验报告

实验题目： Homework 2_1		学号：201900130024
日期：2021.10.11	班级： 数据 19	姓名： 刘士渤
Email：liuburger@qq.com		

实验目的：

掌握基本的神经网络调整技能，并尝试改进深度神经网络:超参数调整、正则化和优化

实验软件和硬件环境：

VScode JupyterNoteBook

联想拯救者 Y7000p

实验原理和方法：

neural network

实验步骤：（不要求罗列完整源代码）

1. 补全 Initialization.ipynb

根据 W_l 的形状是(layers_dims[L], layers_dims[L-1])

b_l 的形状是(layers_dims[L], 1)得代码：

```
parameters['W' + str(l)] = np.zeros([layers_dims[l], layers_dims[l-1]])
parameters['b' + str(l)] = np.zeros([layers_dims[l], 1])
```

根据随机的要求得代码：

```
parameters['W' + str(l)] = np.random.randn(layers_dims[l], layers_dims[l-1])*10
parameters['b' + str(l)] = np.zeros([layers_dims[l], 1])
```

根据提示开方得代码：

```
parameters['W' + str(l)] = np.random.randn(layers_dims[l], layers_dims[l-1])*np.sqrt(2./layers_dims[l-1])
parameters['b' + str(l)] = np.zeros([layers_dims[l], 1])
```

2. 补全 Gradient Checking.ipynb

根据 J(theta) = theta * x 得代码：

```
J = theta*x
```

根据

$$dtheta = \frac{\partial J}{\partial \theta} = x.$$

得代码：

```
dtheta = x
```

根据

1. $\theta^{+} = \theta + \varepsilon$
2. $\theta^{-} = \theta - \varepsilon$
3. $J^{+} = J(\theta^{+})$
4. $J^{-} = J(\theta^{-})$
5. $gradapprox = \frac{J^{+} - J^{-}}{2\varepsilon}$

$$difference = \frac{\| grad - gradapprox \|_2}{\| grad \|_2 + \| gradapprox \|_2}$$

得代码：

```
thetaplus = theta+epsilon                                    # Step
thetaminus = theta-epsilon                                   # Step
J_plus = thetaplus*x                                        # Step
J_minus = thetaminus*x                                      # Step
gradapprox = (J_plus-J_minus)/(2*epsilon)                   # Step
### END CODE HERE ###

# Check if gradapprox is close enough to the output of backwar
### START CODE HERE ### (approx. 1 line)
grad = x
### END CODE HERE ###

### START CODE HERE ### (approx. 1 line)
numerator = np.linalg.norm(grad-gradapprox)
denominator = np.linalg.norm(grad)+np.linalg.norm(gradapprox)
difference = numerator/denominator
```

根据

- To compute `J_plus[i]`:
    1. Set θ^{+} to `np.copy(parameters_values)`
    2. Set θ_i^{+} to $\theta_i^{+} + \varepsilon$
    3. Calculate J_i^{+} using to `forward_propagation_n(x, y, vector_to_dictionary(`θ^{+} `))`.
- To compute `J_minus[i]`: do the same thing with θ^{-}
- Compute $gradapprox[i] = \frac{J_i^{+} - J_i^{-}}{2\varepsilon}$

$$difference = \frac{\| grad - gradapprox \|_2}{\| grad \|_2 + \| gradapprox \|_2}$$

得代码:

```
    thetaplus = np.copy(parameters_values)
    thetaplus[i][0] +=epsilon                                # Step 2
    J_plus[i], _ = forward_propagation_n(X,Y,vector_to_dictionary(thetaplus))
    ### END CODE HERE ###

    # Compute J_minus[i]. Inputs: "parameters_values, epsilon". Output = "J_mini
    ### START CODE HERE ### (approx. 3 lines)
    thetaminus = np.copy(parameters_values)
    thetaminus[i][0] -=epsilon                               # Step 2
    J_minus[i], _ = forward_propagation_n(X,Y,vector_to_dictionary(thetaminus))
    ### END CODE HERE ###

    # Compute gradapprox[i]
    ### START CODE HERE ### (approx. 1 line)
    gradapprox[i] = (J_plus[i]-J_minus[i])/(2*epsilon)
    ### END CODE HERE ###

# Compare gradapprox to backward propagation gradients by computing difference.
### START CODE HERE ### (approx. 1 line)
numerator = np.linalg.norm(grad-gradapprox)                          # Step
denominator = np.linalg.norm(grad)+np.linalg.norm(gradapprox)           #
difference = numerator/denominator                                    # St
```

3. 补全 Optimization methods.ipynb

根据

$$W^{[l]} = W^{[l]} - \alpha \; dW^{[l]}$$

$$b^{[l]} = b^{[l]} - \alpha \; db^{[l]}$$

得代码:

```
parameters["W" + str(l+1)] -= learning_rate*grads['dW' + str(l+1)]
parameters["b" + str(l+1)] -= learning_rate*grads['db' + str(l+1)]
```

根据

```
first_mini_batch_X = shuffled_X[:, 0 : mini_batch_size]
second_mini_batch_X = shuffled_X[:, mini_batch_size : 2 * mini_batch_size]
```

得代码:

```
mini_batch_X = shuffled_X[:,k*mini_batch_size:(k+1)*mini_batch_size]
mini_batch_Y = shuffled_Y[:,k*mini_batch_size:(k+1)*mini_batch_size]
```

根据

$(m - mini\_batch\_size \times \lfloor \frac{m}{mini\_batch\_size} \rfloor)$.

得代码：

```
mini_batch_X = shuffled_X[:,num_complete_minibatches*mini_batch_size:m]
mini_batch_Y = shuffled_Y[:,num_complete_minibatches*mini_batch_size:m]
```

根据

```
v["dW" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["W" + str(l+1)])
v["db" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["b" + str(l+1)])
```

得代码：

```
v["dW" + str(l+1)] = np.zeros([parameters['W' + str(l+1)].shape[0], parameters['W' + str(l+1)].shape[1]])
v["db" + str(l+1)] = np.zeros([parameters['b' + str(l+1)].shape[0], parameters['b' + str(l+1)].shape[1]])
```

根据

$$\begin{cases} v_{dW^{[l]}} = \beta v_{dW^{[l]}} + (1-\beta) dW^{[l]} \\ W^{[l]} = W^{[l]} - \alpha v_{dW^{[l]}} \end{cases}$$

$$\begin{cases} v_{db^{[l]}} = \beta v_{db^{[l]}} + (1-\beta) db^{[l]} \\ b^{[l]} = b^{[l]} - \alpha v_{db^{[l]}} \end{cases}$$

得代码：

```
# compute velocities
v["dW" + str(l+1)] = beta*v["dW" + str(l+1)]+(1-beta)*grads['dW' + str(l+1)]
v["db" + str(l+1)] = beta*v["db" + str(l+1)]+(1-beta)*grads['db' + str(l+1)]
# update parameters
parameters["W" + str(l+1)] -= learning_rate*v["dW" + str(l+1)]
parameters["b" + str(l+1)] -= learning_rate*v["db" + str(l+1)]
```

根据

```
v["dW" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["W" + str(l+1)])
v["db" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["b" + str(l+1)])
s["dW" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["W" + str(l+1)])
s["db" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["b" + str(l+1)])
```

得代码：

```
v["dW" + str(l+1)] = np.zeros([parameters['W' + str(l+1)].shape[0], parameters['W' + str(l+1)].shape[1]])
v["db" + str(l+1)] = np.zeros([parameters['b' + str(l+1)].shape[0], parameters['b' + str(l+1)].shape[1]])
s["dW" + str(l+1)] = np.zeros([parameters['W' + str(l+1)].shape[0], parameters['W' + str(l+1)].shape[1]])
s["db" + str(l+1)] = np.zeros([parameters['b' + str(l+1)].shape[0], parameters['b' + str(l+1)].shape[1]])
```

根据

$$\begin{cases} v_{W^{[l]}} = \beta_1 v_{W^{[l]}} + (1-\beta_1) \frac{\partial J}{\partial W^{[l]}} \\ v^{corrected}_{W^{[l]}} = \frac{v_{W^{[l]}}}{1-(\beta_1)^t} \\ s_{W^{[l]}} = \beta_2 s_{W^{[l]}} + (1-\beta_2) (\frac{\partial J}{\partial W^{[l]}})^2 \\ s^{corrected}_{W^{[l]}} = \frac{s_{W^{[l]}}}{1-(\beta_2)^t} \\ W^{[l]} = W^{[l]} - \alpha \frac{v^{corrected}_{W^{[l]}}}{\sqrt{s^{corrected}_{W^{[l]}}} + \varepsilon} \end{cases}$$

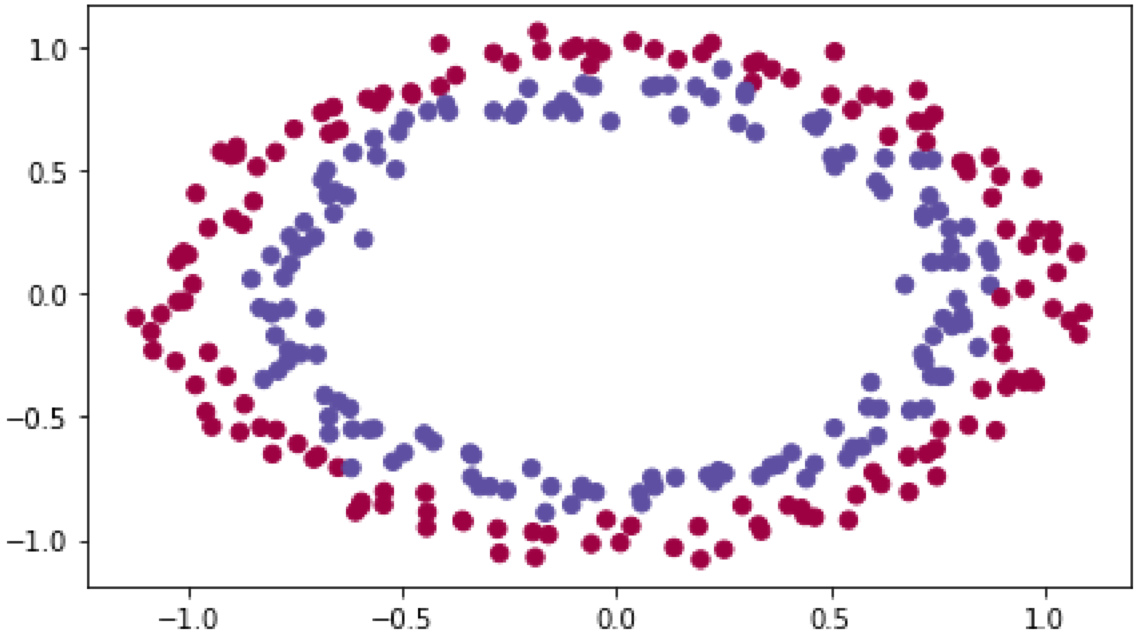
得代码：

```
v["dW" + str(l+1)] = beta1*v["dW" + str(l+1)]+(1-beta1)*grads['dW' + str(l+1)]
v["db" + str(l+1)] = beta1*v["db" + str(l+1)]+(1-beta1)*grads['db' + str(l+1)]
### END CODE HERE ###

# Compute bias-corrected first moment estimate. Inputs: "v, beta1, t". Output: "v_corrected".
### START CODE HERE ### (approx. 2 lines)
v_corrected["dW" + str(l+1)] = v["dW" + str(l+1)]/(1-beta1**t)
v_corrected["db" + str(l+1)] = v["db" + str(l+1)]/(1-beta1**t)
### END CODE HERE ###

# Moving average of the squared gradients. Inputs: "s, grads, beta2". Output: "s".
### START CODE HERE ### (approx. 2 lines)
s["dW" + str(l+1)] = beta2*s["dW" + str(l+1)]+(1-beta2)*(grads['dW' + str(l+1)]**2)
s["db" + str(l+1)] = beta2*s["db" + str(l+1)]+(1-beta2)*(grads['db' + str(l+1)]**2)
### END CODE HERE ###

# Compute bias-corrected second raw moment estimate. Inputs: "s, beta2, t". Output: "s_corrected".
### START CODE HERE ### (approx. 2 lines)
s_corrected["dW" + str(l+1)] = s["dW" + str(l+1)]/(1-beta1**t)
s_corrected["db" + str(l+1)] = s["db" + str(l+1)]/(1-beta1**t)
### END CODE HERE ###

# Update parameters. Inputs: "parameters, learning_rate, v_corrected, s_corrected, epsilon". Output: "parameters".
### START CODE HERE ### (approx. 2 lines)
parameters["W" + str(l+1)] -= learning_rate*v_corrected["dW" + str(l+1)]/(np.sqrt(s_corrected["dW" + str(l+1)])+epsilon)
parameters["b" + str(l+1)] -= learning_rate*v_corrected["db" + str(l+1)]/(np.sqrt(s_corrected["db" + str(l+1)])+epsilon)
```

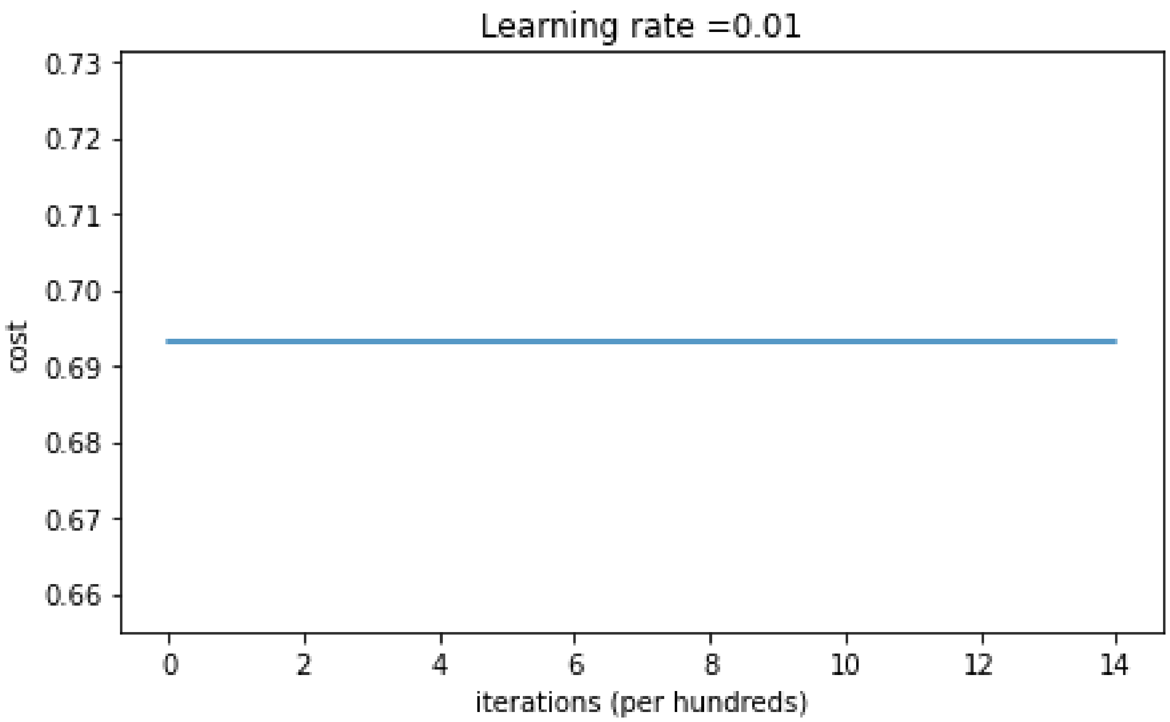
结论分析与体会：

1. Initialization

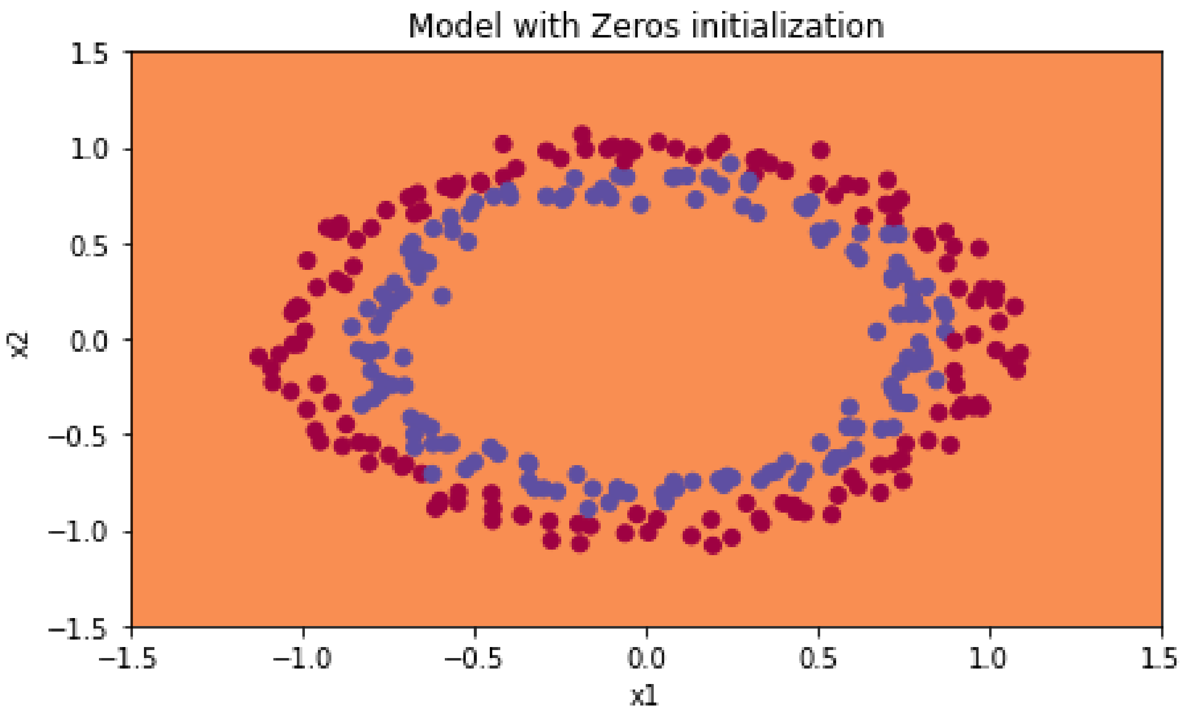
原数据：



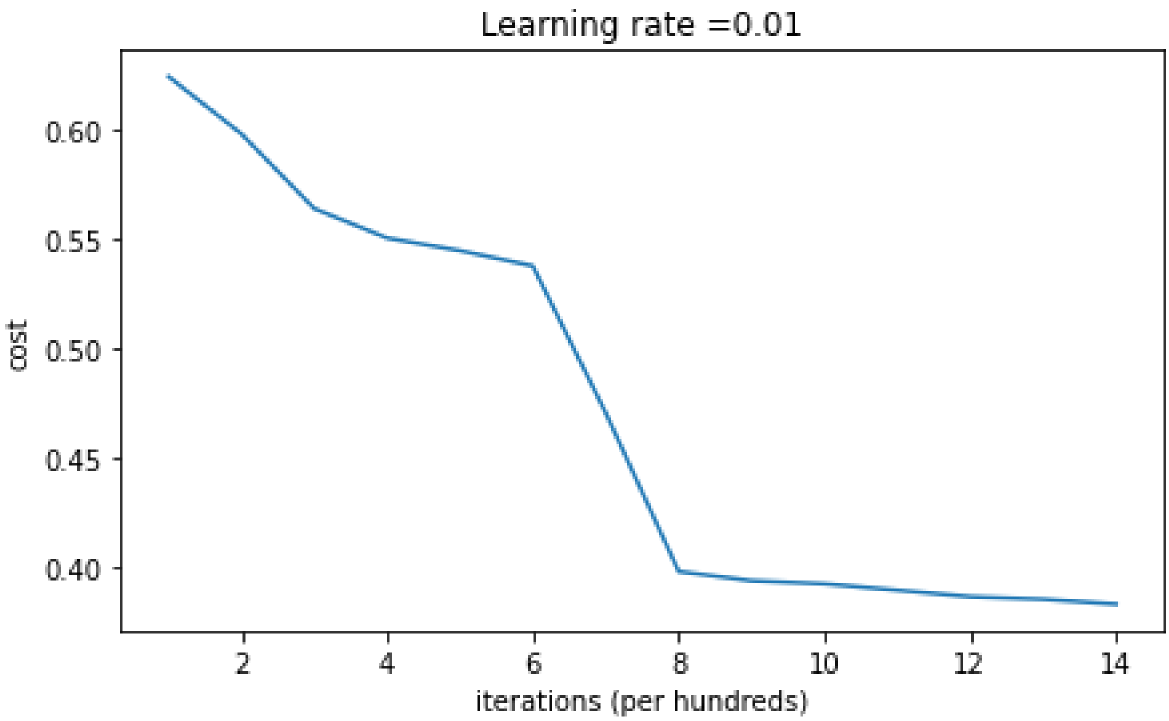
采用全 0 初始化的 cost 与 iterations 的关系（因为初始化为 0 所以迭代没有任何效果）：



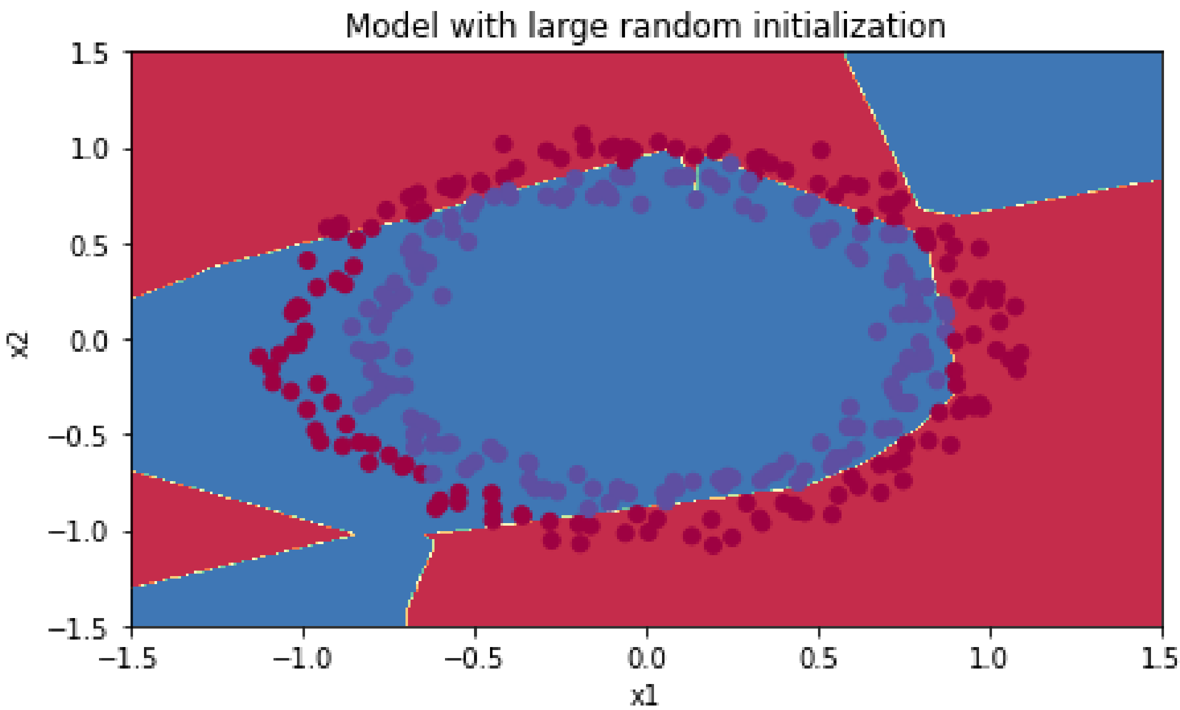
预测准确率为 0.5：



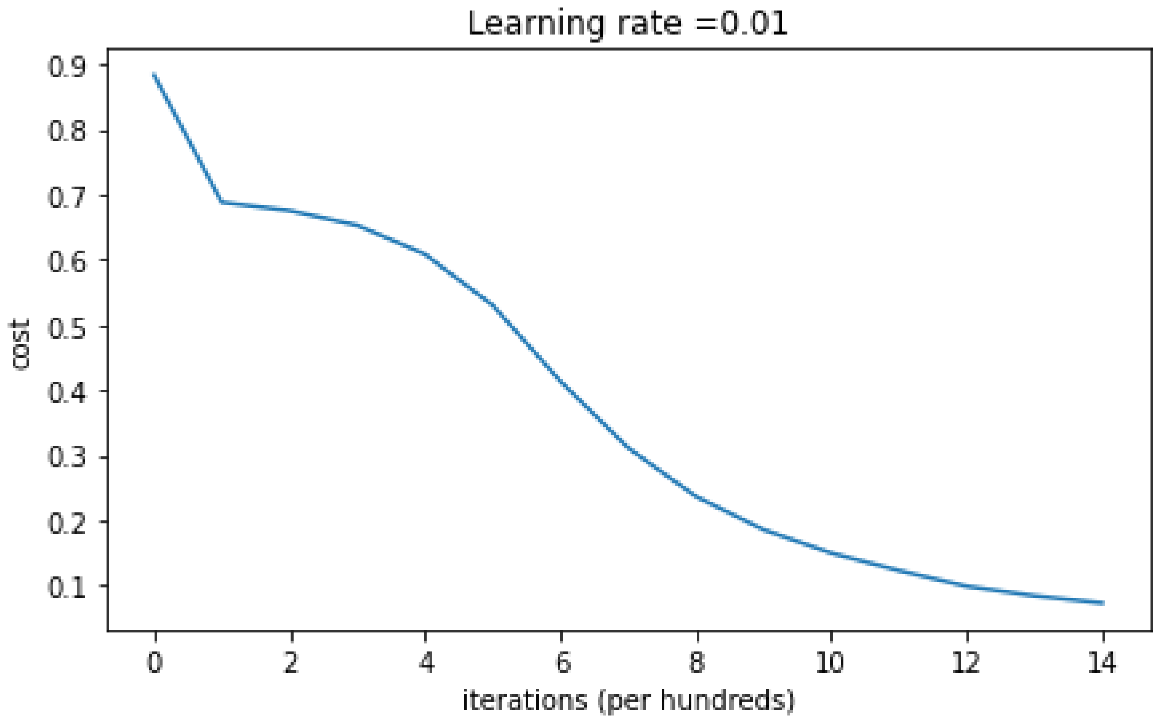
采用随机数初始化的 cost 与 iterations 的关系（损失函数随迭代次数增加而降低，速度为快->慢->快->慢）：



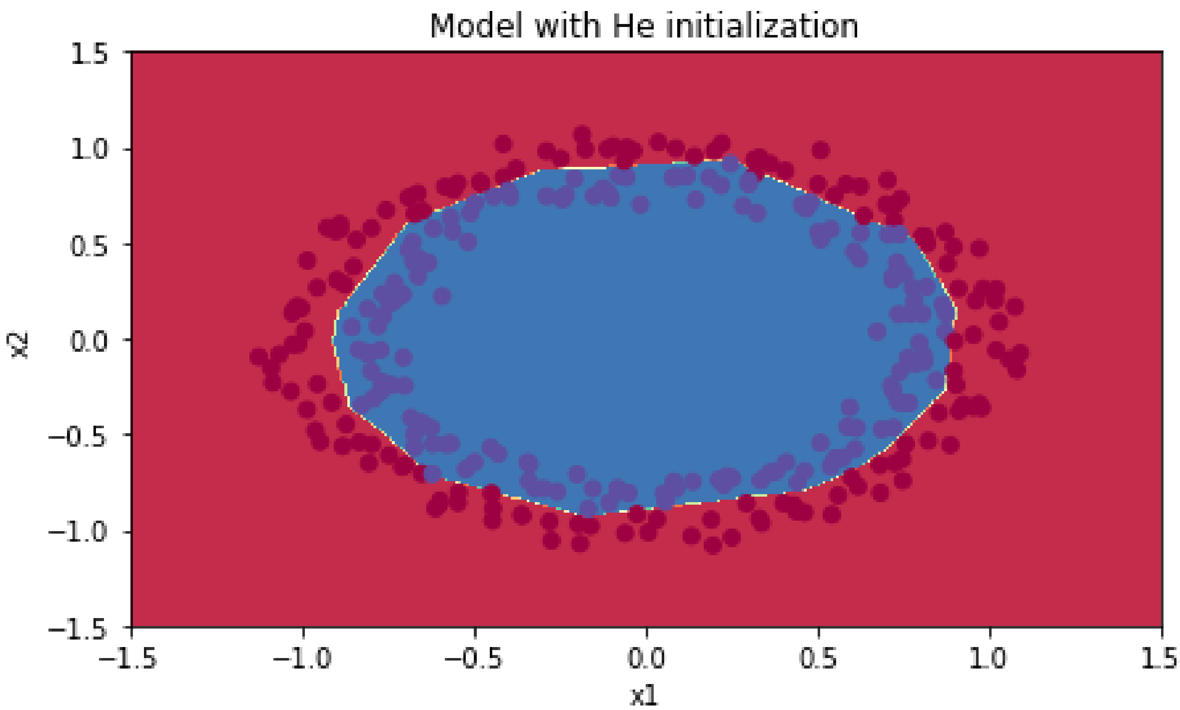
预测准确率为 0.83



采用 he 初始化的 cost 与 iterations 的关系（损失函数随迭代次数增加而降低，速度同样为快->慢->快->慢，但较随机初始化更为平滑）：



预测准确率为 0.99



2. Gradient Checking

写了梯度估计后，与梯度做差，发现差异很小：

```
The gradient is correct!
difference = 2.919335883291695e-10
```

修改了错误代码之后

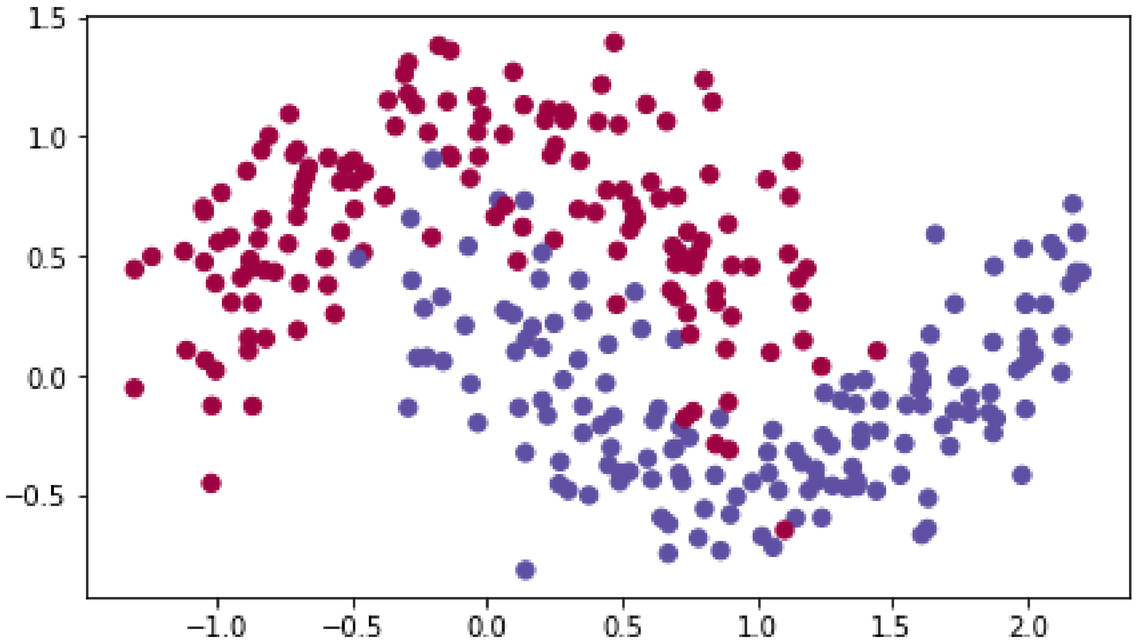
```
dW2 = 1./m * np.dot(dZ2, A1.T)
db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)

dA1 = np.dot(W2.T, dZ2)
dZ1 = np.multiply(dA1, np.int64(A1 > 0))
dW1 = 1./m * np.dot(dZ1, X.T)
db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)
```

差异由 0.29 变为 1.19e-7

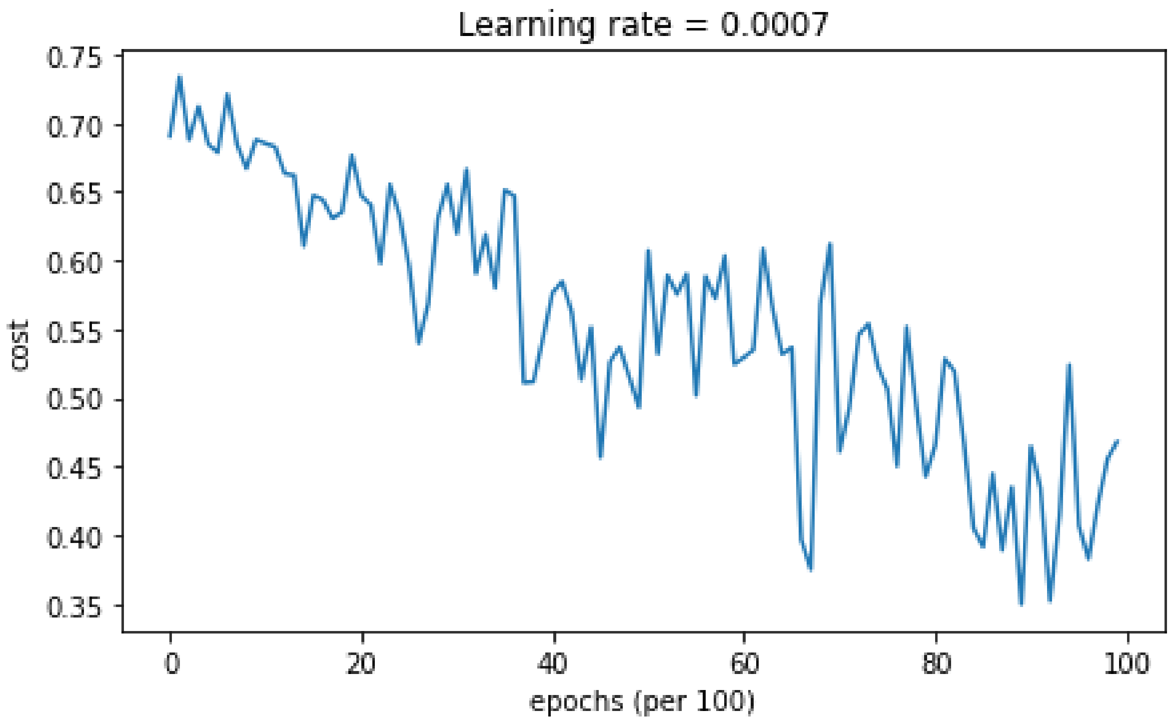
```
Your backward propagation works perfectly fine! difference = 1.1890913024229996e-07
```

3. Optimization methods

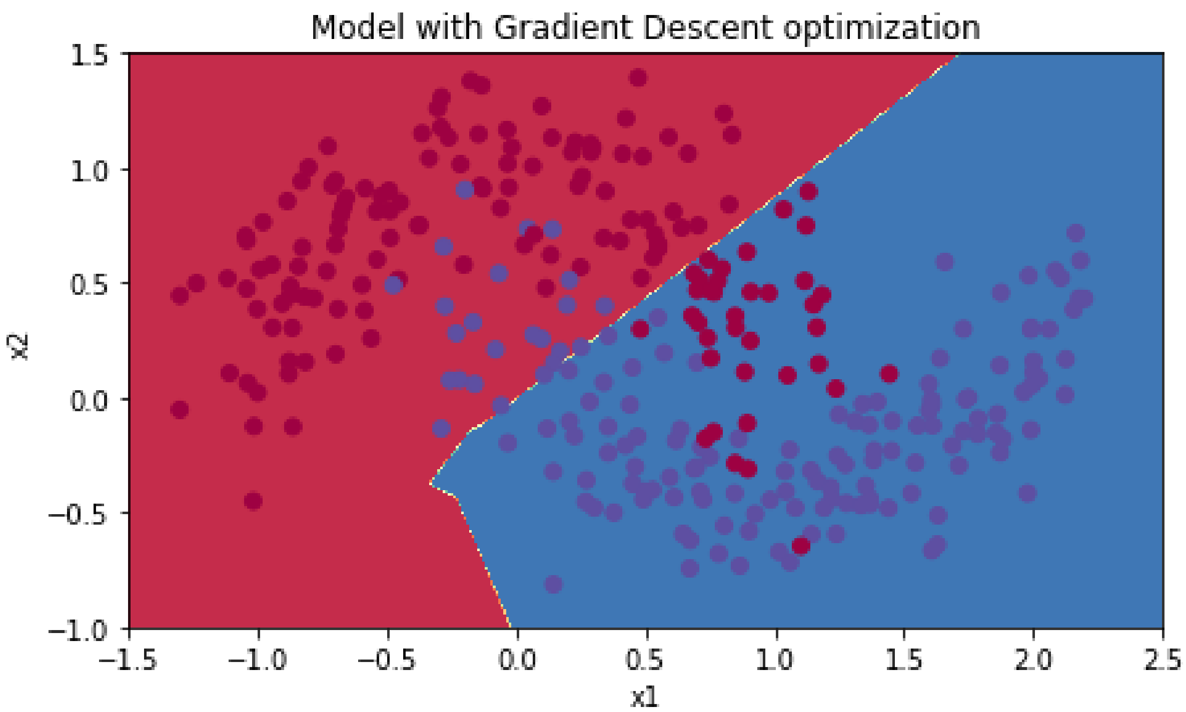
原数据



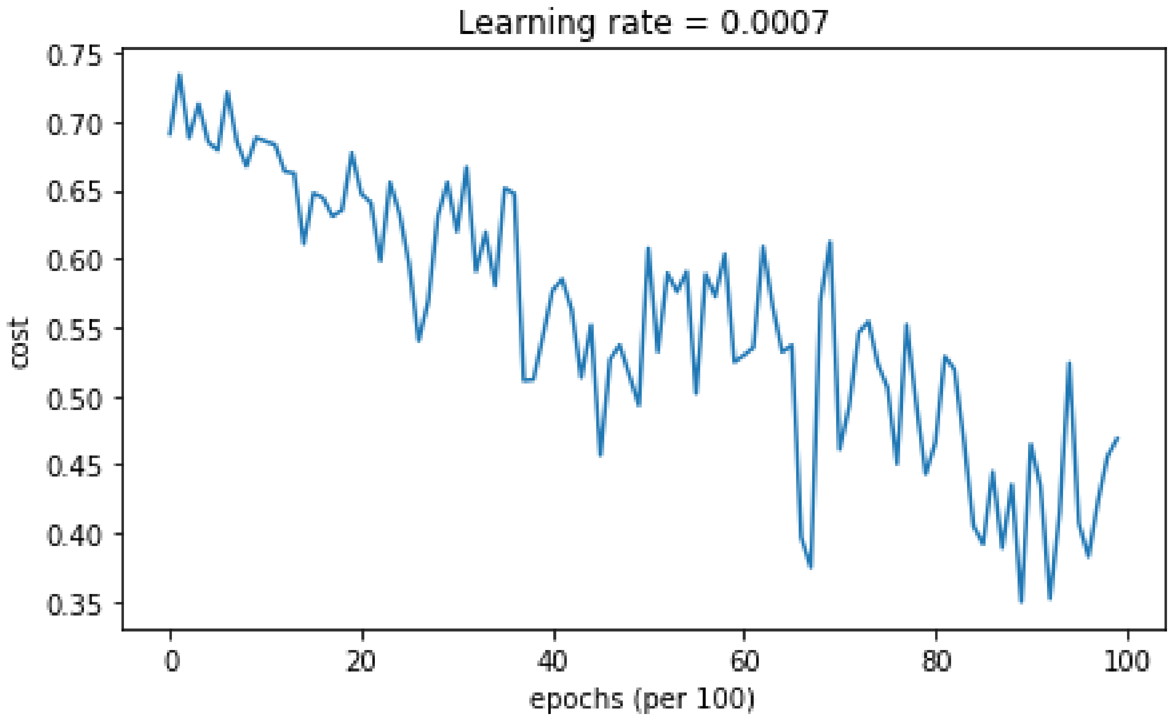
采用梯度下降方法的 cost 与 epochs 的关系（损失函数随迭代次数增加而降低，总体呈下降趋势但还没有趋于定值）：



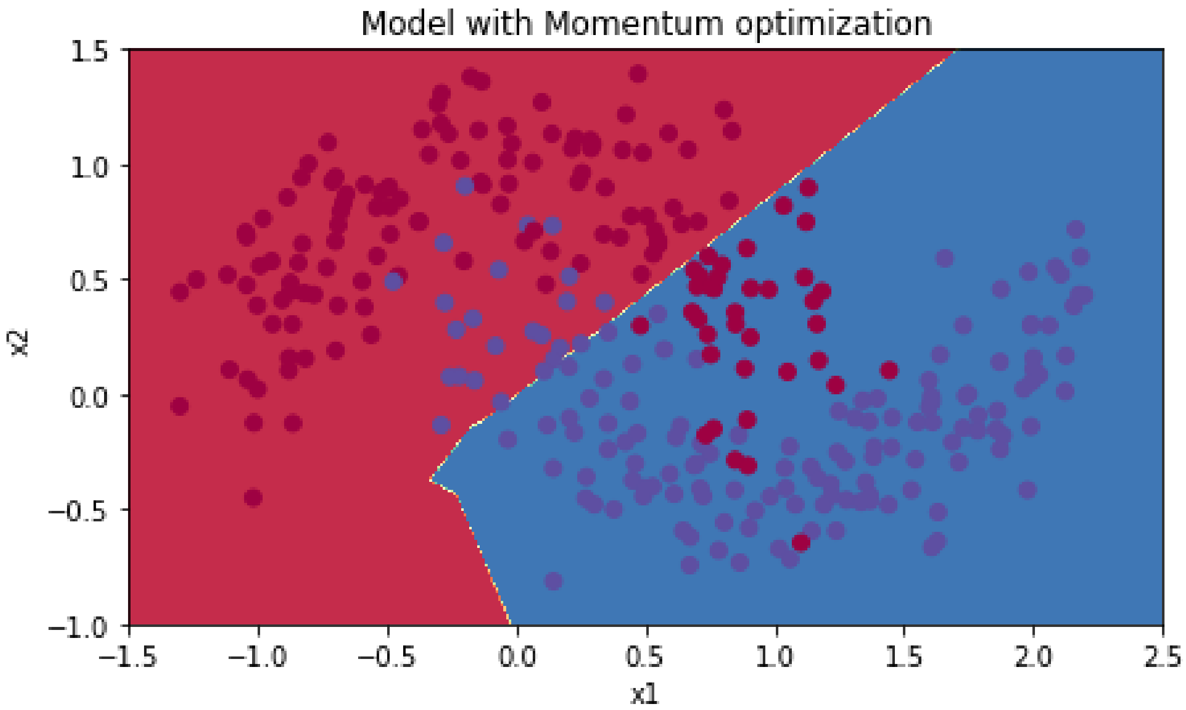
预测准确率为 0.79



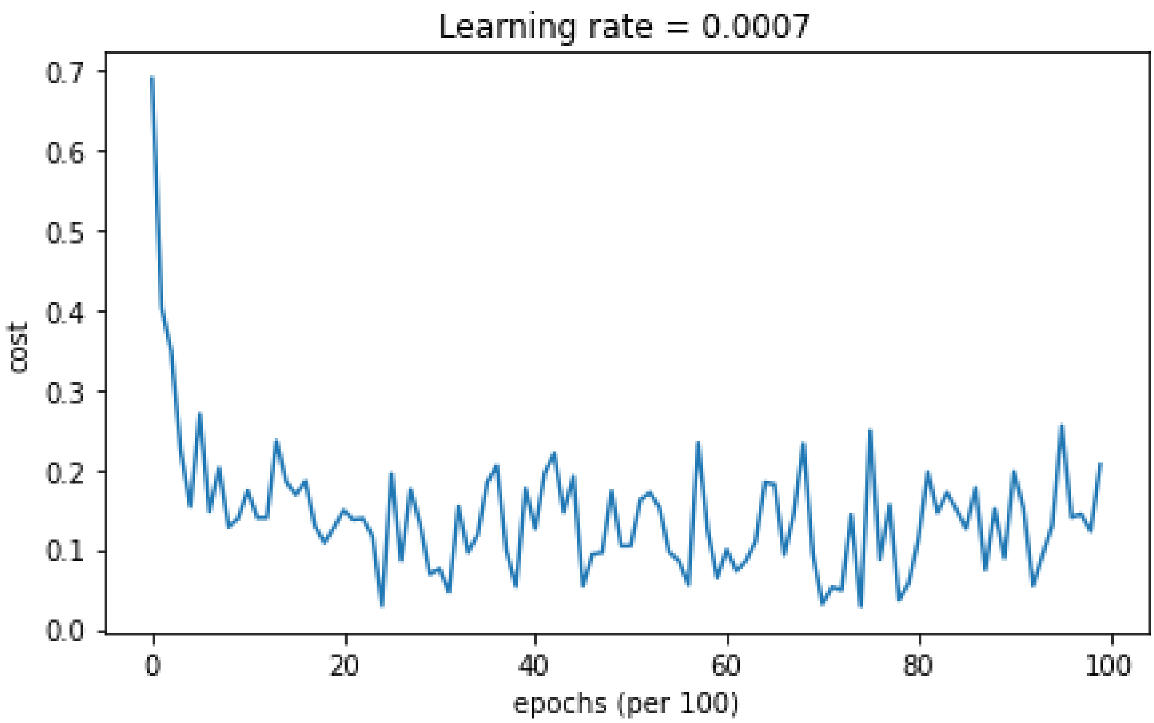
采用动量方法的 cost 与 epochs 的关系（图像似乎和梯度下降的一模一样）：



预测准确率为 0.79



采用动量方法的 cost 与 epochs 的关系（损失函数随迭代次数增加而降低，最终稳定在 0.15 左右）：



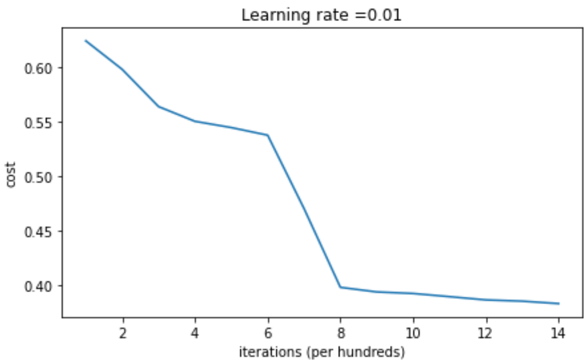
预测准确率为 0.94

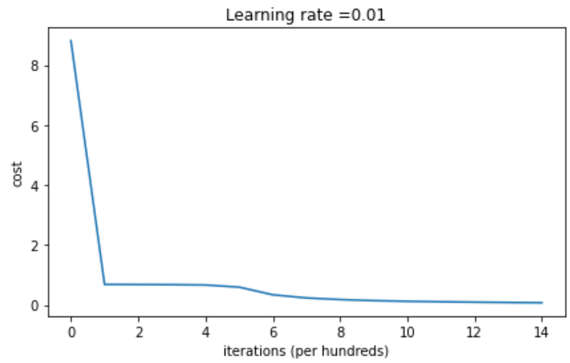
就实验过程中遇到和出现的问题，你是如何解决和处理的，自拟 1—3 道问答题：

1. Initialization 部分，我尝试着提升 random 初始化的准确率。在将 W 的系数从 10 改为 1.7 之后

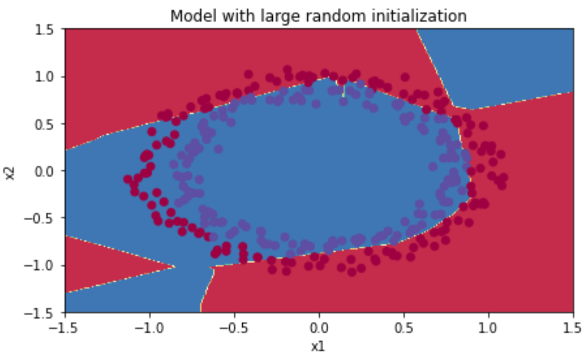
```
parameters['W' + str(l)] = np.random.randn(layers_dims[l], layers_dims[l-1])*1.7
```

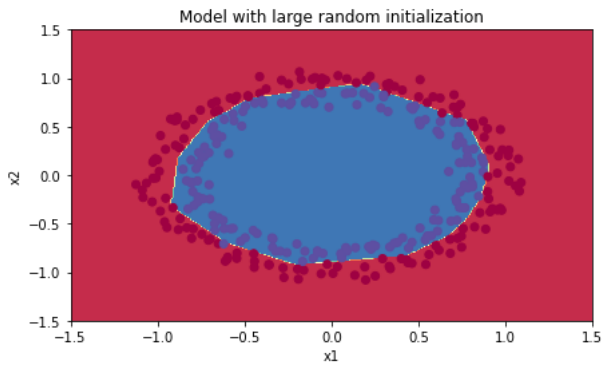
损失函数随迭代较之前收敛更快：





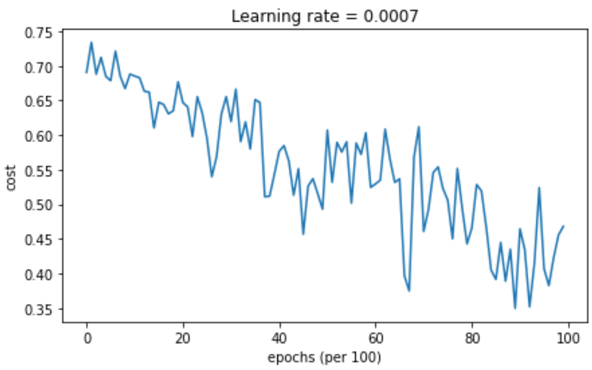
预测准确率也提高到了 0.99

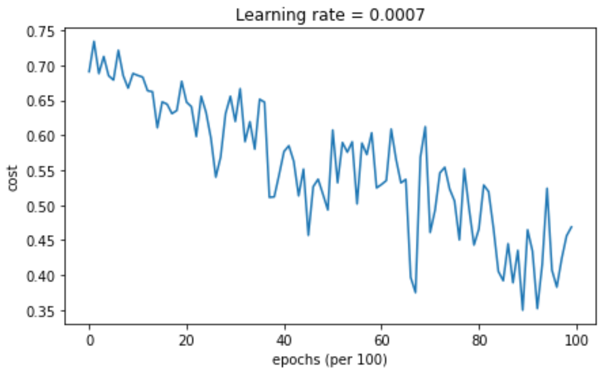




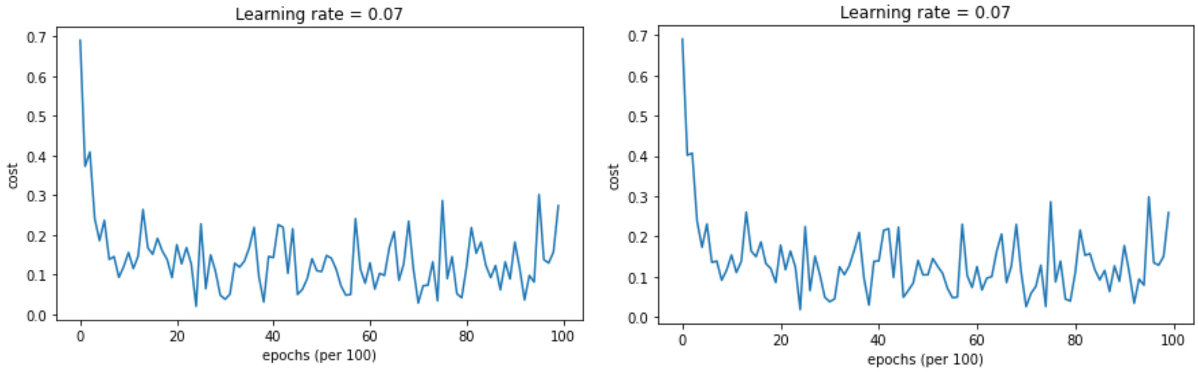
只是改了一下系数，却达到了 he 方法的准确率；看来 1.7 这个数值和对 2/前一层的维度 开方是差不多的。

2. Optimization 部分，由于观察到采用梯度下降和动量方法的损失函数曲线几乎一模一样，并且波动较大，趋于定值的趋势不明显，所以我决定题高 learning rate。

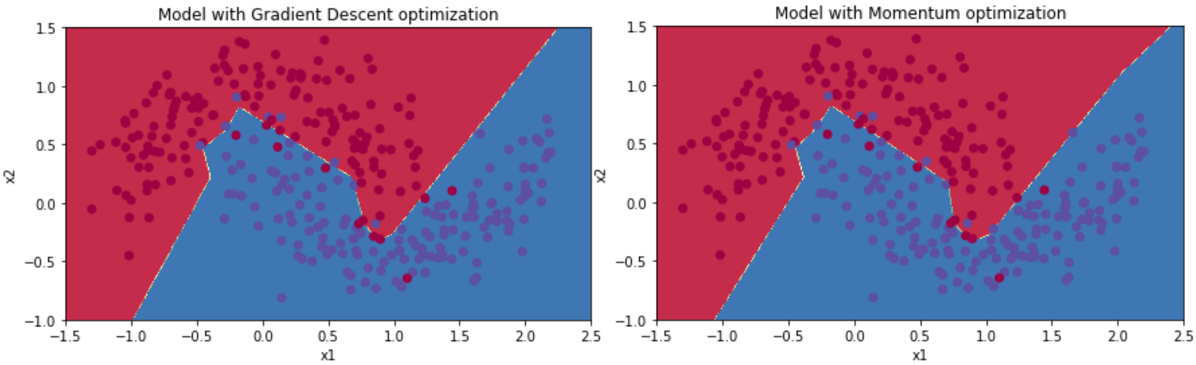




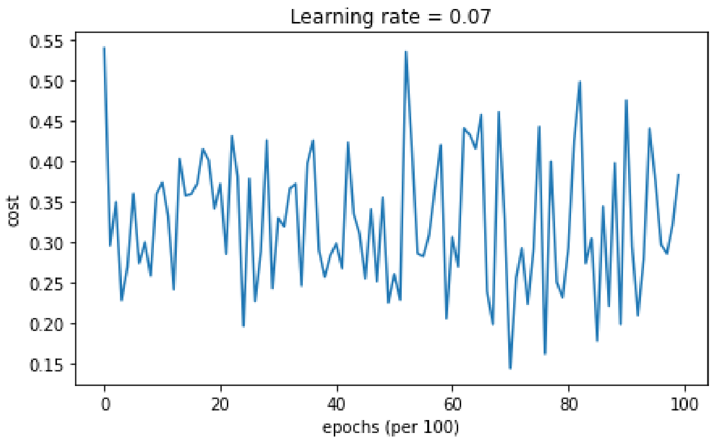
将 learning rate 题高到 100 倍即 0.07 后，损失函数就像 Adam 一样收敛了。



预测准确率提高到了 0.94



但是 Adam 的损失函数却波动起来了



且准确率也降低到 0.87，看来梯度下降和动量方法适合较大的学习率，而 Adam 方法适合较小的学习率。