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：神经网络的调参		学号：202020130190
日期：2022.10.02	班级： 人工智能	姓名：刘绪波
Email：2842353032@qq.com		

实验目的：

通过本次实验，需要掌握基本的神经网络的调参技能，并学习如何对神经网络进行改进，如超参数调整，梯度检查，优化等

实验软件和硬件环境：

软件: Dataspell 2022.2.2 ；

硬件：CPU：AMD Ryzen 7 4800U; 显卡：AMD Radeon

实验原理和方法：

1． 对于权重的初始化：

为了解决深度神经网络产生梯度消失和梯度爆炸的问题，可以采用为神经网络更谨慎地选择随机初始化参数。除此之外，初始化还对模型的收敛速度和性能有着至关重要的影响，神经网络其实就是对权重参数 w 的不停迭代更新，以期达到较好的性能；

目前常用的方法：全 0 初始化，随机初始化，he 初始化

① 全 0 初始化(Zero initialization)：全部参数都初始化为 0；

② 随机初始化(Random initialization): 初始化为随机数(没有控制大小);

③ He 初始化(He initialization): 随机初始化了之后，再利用一个公式，这样就避免了参数的初始值过大或者过小，因此可以取得比较好的效果

2． 梯度检验：

通常梯度检验用来检查已经实现的 bp 是否正确。其实所谓梯度检验，就是自己实现下导数的定义，去求 w 和 b 的导数（梯度），然后去和 bp 求到的梯度比较，如果差值在很小的范围内，则可以认为我们实现的 bp 没问题。那么具体如何工作的呢?

Backpropagation 计算梯度(the gradients)J 对 theta 的偏导数，θ 代表着模型的参数，J 是使用前向传播和 loss function 来计算的。定义函数：

$$\frac{\partial J}{\partial \theta} = \lim_{\varepsilon \to 0} \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2\varepsilon}$$

我们利用 theta 和一个很小的 epsilon，在代码中使用 epsilon = 1e -7;

3． 优化：

之所以进行优化就是为了提升参数和最小化成本，通过此部分学习，可以了解到一般常用的除了梯度下降之外的一些高级方法来进行优化，比如一个好的优化算法可以是我们预测或者分类的更加准确，提高计算速度，减小运算成本等；主要介绍了 BGD，SGD，**Mini-Batch GD，momentum ，adam**

实验步骤：（不要求罗列完整源代码）

首先，启动 IPython，打开文件夹作业二，找到对应的 ipython 文件：阅读补全代码：

1. 对于权重的初始化：

① 对于全 0 权重的初始化：

首先我们需要对代码进行修改，利用 np.zeros（）来初始化，并且要使用正确的维度；修改代码的方式如下：

```
    parameters = {}
    L = len(layers_dims)            # number of layers in the network

    for l in range(1, L):
        ### START CODE HERE ### (≈ 2 lines of code)
        parameters['W' + str(l)] = np.zeros((layers_dims[l], layers_dims[l-1]))
        parameters['b' + str(l)] = np.zeros((layers_dims[l],1))
        ### END CODE HERE ###
    return parameters
```

② 对于随机初始化：

实现函数将权重初始化为较大的随机值（按 *10 缩放）并将偏差初始化为零。实现代码如下：

```
L = len(layers_dims)            # integer representing the number of layers

for l in range(1, L):
    ### START CODE HERE ### (≈ 2 lines of code)
    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))
    ### END CODE HERE ###

return parameters
```

③ 对于 He 初始化：

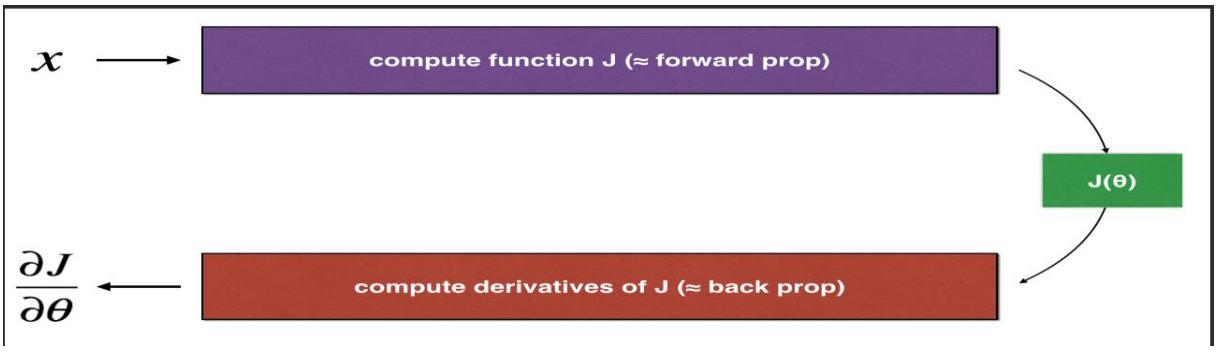
先对权重进行随机初始化，只是将上面的按 10 倍缩放变化为用另一个公式缩放，代码如下：

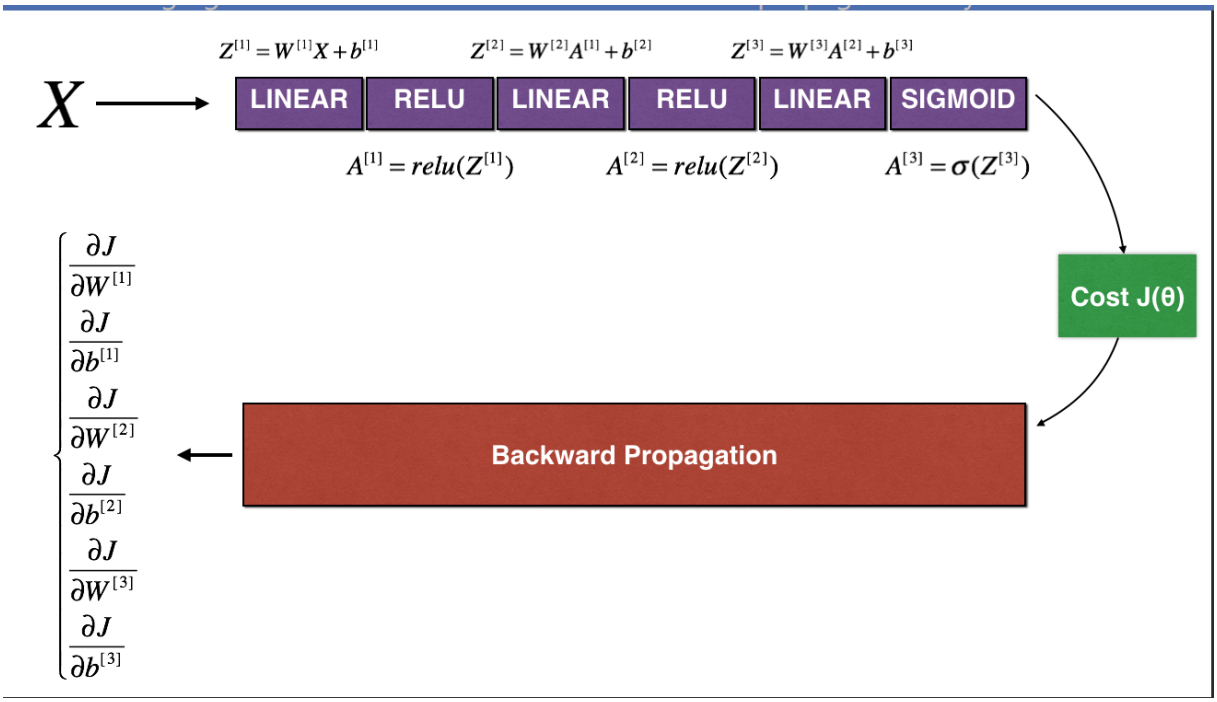
```
L = len(layers_dims) - 1 # integer representing the number of layers

for l in range(1, L + 1):
    ### START CODE HERE ### (≈ 2 lines of code)
    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))
    ### END CODE HERE ###
```

2. 梯度检验：

首先根据下图计算过程来完成步骤：首先始于 x，然后评估 J(x)（forward prop），然后计算偏导数（back prop）；





① 首先我们考虑一维的梯度检验：

实现简单函数的 forward prop 和 backward prop:

```
    ### START CODE HERE ### (approx. 1 line)
    J = np.dot(theta,x)
    ### END CODE HERE ###

    return J
```

```
x, theta = 2, 4
J = forward_propagation(x, theta)
print ("J = " + str(J))
```

```
J = 8
```

Expected Output:

** J **	8

```
    ### START CODE HERE ### (approx. 1 line)
    dtheta = x
    ### END CODE HERE ###

    return dtheta
```

```
x, theta = 2, 4
dtheta = backward_propagation(x, theta)
print ("dtheta = " + str(dtheta))
```

```
dtheta = 2
```

Expected Output:

** dtheta **	2

以上两个分别为 forward prop 和 backward prop 并且可以发现输出结果和预期结果一致；

接下来为了证明我们的 backward_propagation()的正确性，采用以下方法：

计算“gradapprox”：公式和代码如下：

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

```
# Compute gradapprox using left side of formula (1). epsilon is sma
  to worry about the limit.
### START CODE HERE ### (approx. 5 lines)
thetaplus = theta + epsilon                                # Step 1
thetaminus = theta - epsilon                               # Step 2
J_plus = forward_propagation(x,thetaplus)                  # Step 3
J_minus = forward_propagation(x,thetaminus)                # Step 4
gradapprox = (J_plus-J_minus) / (2 * epsilon)              # Step 5
```

然后使用 backward propagation 计算 gradient 并且将差异记录在变量 grad 中，最后计算“grad”和”gradapprox”之间的差异，公式和代码如下：

$$difference = \frac{|| grad - gradapprox ||_2}{|| grad ||_2 + || gradapprox ||_2} \boxed{2}$$

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

### START CODE HERE ### (approx. 1 line)
numerator = np.linalg.norm(grad - gradapprox)                                  # S
denominator = np.linalg.norm(grad) + np.linalg.norm(gradapprox)                # S
difference = numerator / denominator                                # Step 3'
### END CODE HERE ###
```

② 多维梯度检验：

下面根据上面展示的多维欺诈模型的实现过程的展示：

首先还是对于函数的 forward prop 和 backward prop 的实现；鉴于源代码中已经给出，此处不再粘贴代码：接下来看如何实现梯度检验；根据一维的情况：我们公式不变：仍然是：

$$\frac{\partial J}{\partial \theta} = \lim_{\varepsilon \to 0} \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2\varepsilon}$$

但是此处的 theta 不再是标量，而是一个叫做“parameters”的字典；我们要用字典转向量：

关于具体实现 gradient_check_n()，伪代码及和代码如下：

For each i in num_parameters:

- 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 $gradapprox[i] = \dfrac{J_i^+ - J_i^-}{2\varepsilon}$

Thus, you get a vector gradapprox, where gradapprox[i] is an approximation of the gradient with respect to `parameter_values[i]`. You can now compare this gradapprox vector to the gradients vector from backpropagation. Just like for the 1D case (Steps 1', 2', 3'), compute:

$$difference = \frac{\|grad - gradapprox\|_2}{\|grad\|_2 + \|gradapprox\|_2} \boxed{3}$$

```
### START CODE HERE ### (approx. 3 lines)
thetaplus = np.copy(parameters_values)                                      # Step 1
thetaplus[i][0] = thetaplus[i][0] + epsilon                                 # Step 2
J_plus[i], _ = forward_propagation_n(X,Y,vector_to_dictionary(thetaplus))
                    # Step 3
### END CODE HERE ###

# Compute J_minus[i]. Inputs: "parameters_values, epsilon". Output = "J_minus[i]".
### START CODE HERE ### (approx. 3 lines)
thetaminus = np.copy(parameters_values)                                      # Step 1
thetaminus[i][0] = thetaminus[i][0] - epsilon                                # Step 2
J_minus[i], _ = forward_propagation_n(X,Y,vector_to_dictionary(thetaminus))
                     # Step 3
### 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 1'
denominator = np.linalg.norm (grad) + np.linalg.norm(gradapprox)             # Step 2'
difference = numerator / denominator                                         # Step 3'
### END CODE HERE ###
```

3. **优化：**

① GD and SGD：

公式和代码：

$$W^{[l]} = W^{[l]} - \alpha\ dW^{[l]} \boxed{1}$$

$$b^{[l]} = b^{[l]} - \alpha\ db^{[l]} \boxed{2}$$

```
L = len(parameters) // 2 # number of layers in the neural networks

# Update rule for each parameter
for l in range(L):
    ### START CODE HERE ### (approx. 2 lines)
    parameters["W" + str(l+1)] = parameters["W"+ str(l+1)] - learning_rate * grads["dW" +
     str(l+1)]
    parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - learning_rate * grads["db"+
     str(l+1)]
    ### END CODE HERE ###

return parameters
```

L 是层数的迭代，alpha 是学习率；所有的结果储存在 parameters 的字典中；

② Mini-Batch Gradient descent：

实现 random_mini_batches，代码如下：

```
    for k in range(0, num_complete_minibatches):
        ### START CODE HERE ### (approx. 2 lines)
        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]
        ### END CODE HERE ###
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    # Handling the end case (last mini-batch < mini_batch_size)
    if m % mini_batch_size != 0:
        ### START CODE HERE ### (approx. 2 lines)
        mini_batch_X = shuffled_X[:,num_complete_minibatches * mini_batch_size]
        mini_batch_Y = shuffled_Y[:,num_complete_minibatches * mini_batch_size]
        ### END CODE HERE ###
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)
```

③ Momentum：

小批量梯度下降和随机梯度下降都是震荡型梯度下降，而 GD 采用平滑的收敛；因此采用 Momentum 考虑到了震荡的情况减少了小批量梯度下降是震荡现象的发生；将先前梯度的“方向”存储在变量 中。 形式上，这将是先前步骤的梯度的指数加权平均值。

初始化速度 v。速度 ，是一个 Python 字典，需要用零数组进行初始化：

```
    # Initialize velocity
    for l in range(L):
        ### START CODE HERE ### (approx. 2 lines)
        v["dW" + str(l+1)] = np.zeros_like(parameters["W" + str(l+1)])
        v["db" + str(l+1)] = np.zeros_like(parameters["b" + str(l+1)])
        ### END CODE HERE ###
```

使用 Momentum 实现参数更新：

$$\begin{cases} v_{dW^{[l]}} = \beta v_{dW^{[l]}} + (1-\beta)dW^{[l]} \\ W^{[l]} = W^{[l]} - \alpha v_{dW^{[l]}} \end{cases} \boxed{3}$$

$$\begin{cases} v_{db^{[l]}} = \beta v_{db^{[l]}} + (1-\beta)db^{[l]} \\ b^{[l]} = b^{[l]} - \alpha v_{db^{[l]}} \end{cases} \boxed{4}$$

```
for l in range(L):

    ### START CODE HERE ### (approx. 4 lines)
    # 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)] = parameters["W" + str(l+1)] - learning_rate * v["dW" +
     str(l+1)]
    parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - learning_rate * v["db" +
     str(l+1)]
    ### END CODE HERE ###
```

④ Adam:

Adam 是训练神经网络最有效的优化算法之一。 它结合了 RMSProp 和 Momentum 的想法：

初始化公式和代码：

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

```
# Initialize v, s. Input: "parameters". Outputs: "v, s".
for l in range(L):
### START CODE HERE ### (approx. 4 lines)
    v["dW" + str(l+1)] = np.zeros_like(parameters["W" + str(l+1)])
    v["db" + str(l+1)] = np.zeros_like(parameters["b" + str(l+1)])
    s["dW" + str(l+1)] = np.zeros_like(parameters["W" + str(l+1)])
    s["db" + str(l+1)] = np.zeros_like(parameters["b" + str(l+1)])
### END CODE HERE ###
```

正式训练公式代码：

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

```
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 - np.power(beta1, t))
v_corrected["db" + str(l+1)] = v["db" + str(l + 1)] / (1 - np.power(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) * np.power(grads['dW' + str(l + 1)], 2)
s["db" + str(l+1)] = beta2 * s["db" + str(l + 1)] + (1 - beta2) * np.power(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 - np.power(beta2, t))
s_corrected["db" + str(l+1)] = s["db" + str(l + 1)] / (1 - np.power(beta2, t))
### END CODE HERE ###
```

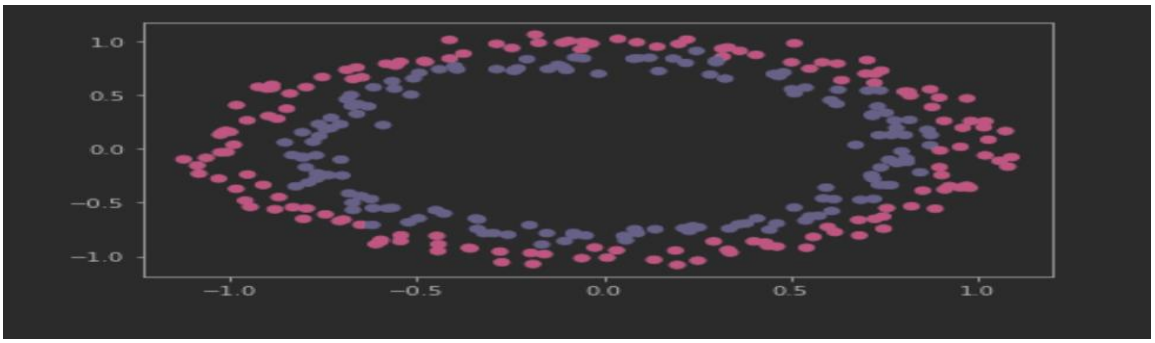
```
# Update parameters. Inputs: "parameters, learning_rate, v_corrected, s_corrected, epsilon". Output: "parameters".
### START CODE HERE ### (approx. 2 lines)
parameters["W" + str(l+1)] = parameters["W" + str(l + 1)] - learning_rate * v_corrected["dW" + str(l + 1)] / np.sqrt(s_corrected["dW" + str(l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

⑤ Model:

使用以下“moons”数据集来测试不同的优化方法。

结论分析与体会：

原始数据情况：



1. 对于权重的初始化：

① 全 0 初始化：

```
W1 = [[0. 0. 0.]
 [0. 0. 0.]]
b1 = [[0.]
 [0.]]
W2 = [[0. 0.]]
b2 = [[0.]]
```

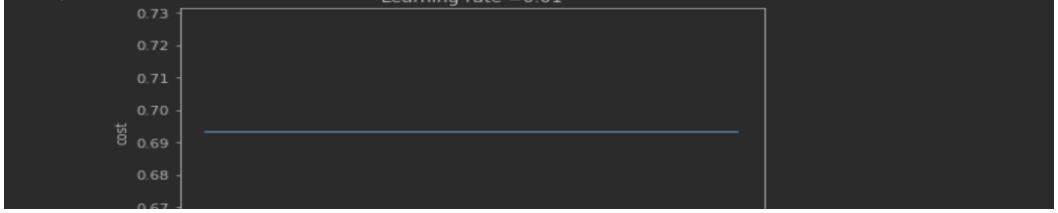
Expected Output:

W1	[[0. 0. 0.] [0. 0. 0.]]
b1	[[0.] [0.]]
W2	[[0. 0.]]
b2	[[0.]]

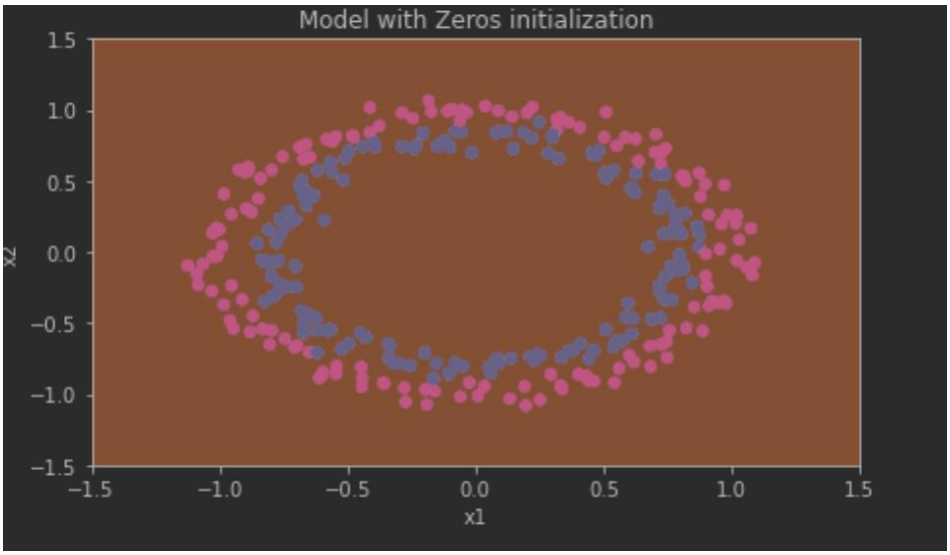
与预期结果一致；

迭代结果，参数变化如下：

```
Cost after iteration 5000: 0.6931471805599453
Cost after iteration 6000: 0.6931471805599453
Cost after iteration 7000: 0.6931471805599453
Cost after iteration 8000: 0.6931471805599453
Cost after iteration 9000: 0.6931471805599453
Cost after iteration 10000: 0.6931471805599455
Cost after iteration 11000: 0.6931471805599453
Cost after iteration 12000: 0.6931471805599453
Cost after iteration 13000: 0.6931471805599453
Cost after iteration 14000: 0.6931471805599453
```



迭代结果从一开始就没有什么变化，因为全零初始化后，神经网络训练时，在反向传播时梯度相同，参数更新也一样，最后会出现输出层两个权值相同，隐层神经元参数相同，也就是说神经网络**失去**了特征学习的能力。



② 随机初始化：

结果展示：

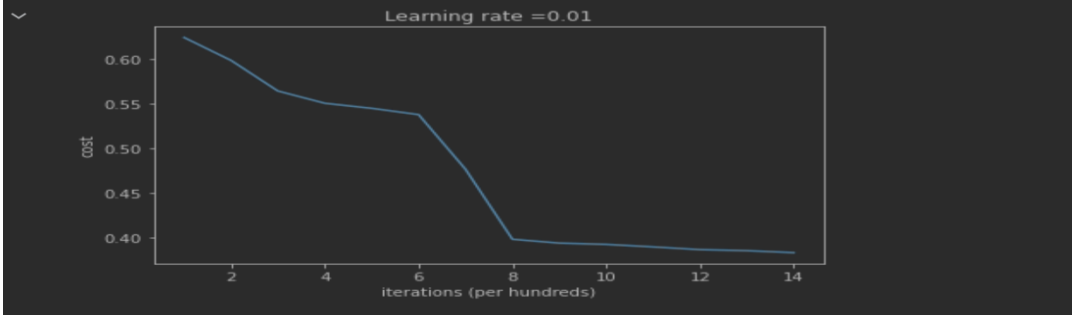
```
W1 = [[ 17.88628473   4.36509851   0.96497468]
 [-18.63492703  -2.77388203  -3.54758979]]
b1 = [[0.]
 [0.]]
W2 = [[-0.82741481 -6.27000677]]
b2 = [[0.]]
```

Expected Output:

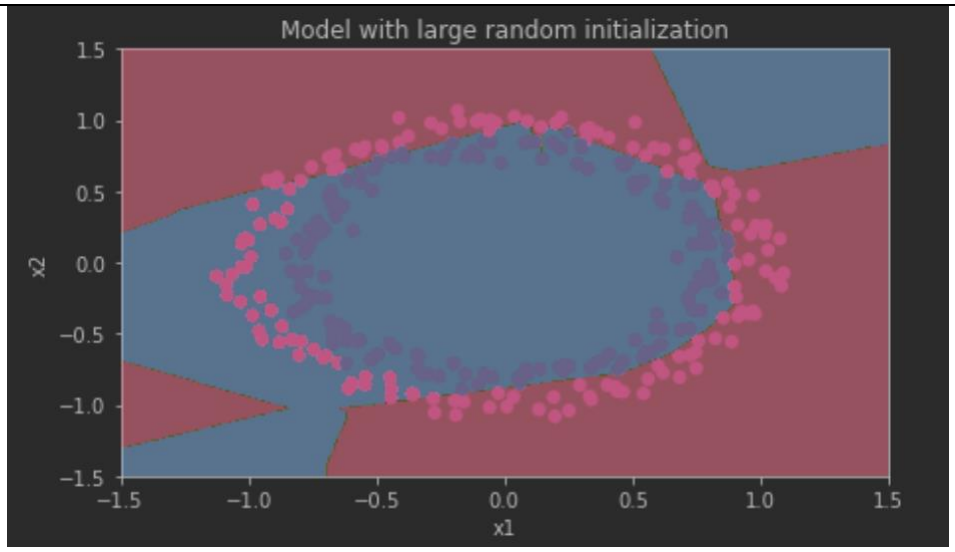
W1	[[17.88628473 4.36509851 0.96497468] [-18.63492703 -2.77388203 -3.54758979]]
b1	[[0.] [0.]]
W2	[[-0.82741481 -6.27000677]]
b2	[[0.]]

与预期结果一致；

```
Cost after iteration 5000: 0.5444639156233223
Cost after iteration 6000: 0.5374519500769401
Cost after iteration 7000: 0.47622027707637854
Cost after iteration 8000: 0.3978137903579155
Cost after iteration 9000: 0.39347526453416565
Cost after iteration 10000: 0.3920300877227725
Cost after iteration 11000: 0.3892619210843658
Cost after iteration 12000: 0.38615660622492826
Cost after iteration 13000: 0.38498645222060723
Cost after iteration 14000: 0.38278526278727415
```



迭代过程也不断再收敛，不断朝着一个更好的方向发展；对于预测的正确率也由原来的 50%上升到了 83%；



③ He 初始化：

结果展示如下：

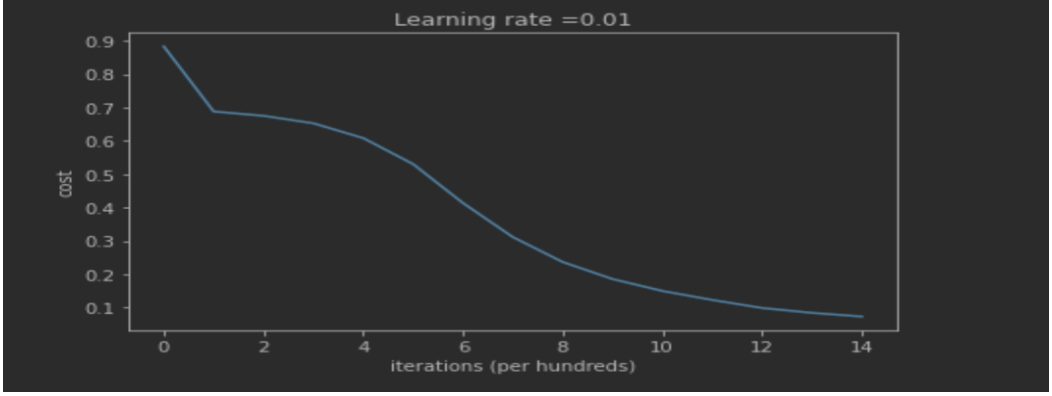
```
W1 = [[ 1.78862847  0.43650985]
 [ 0.09649747 -1.8634927 ]
 [-0.2773882  -0.35475898]
 [-0.08274148 -0.62700068]]
b1 = [[0.]
 [0.]
 [0.]
 [0.]]
W2 = [[-0.03098412 -0.33744411 -0.92904268  0.62552248]]
b2 = [[0.]]
```

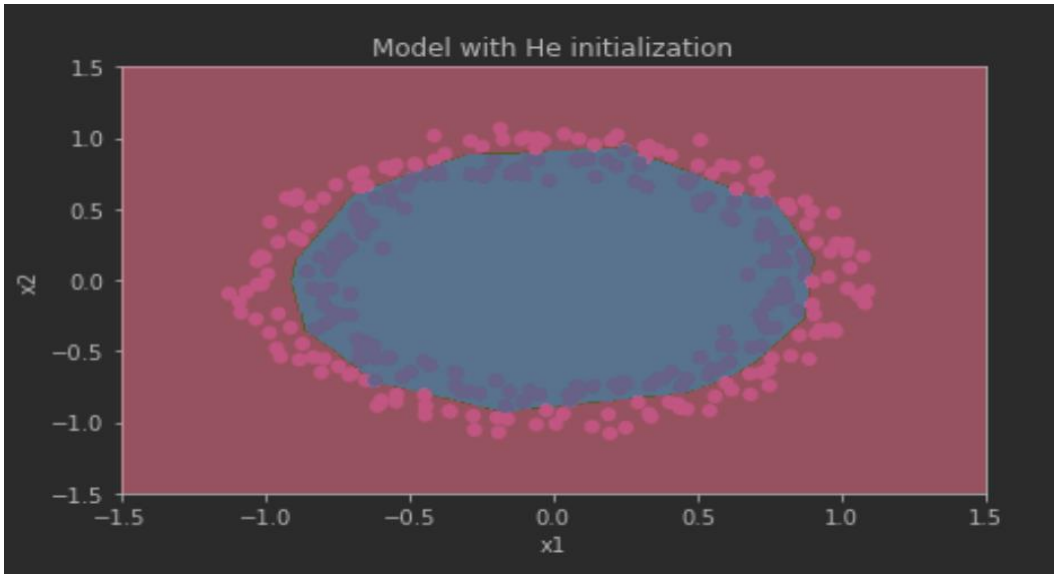
Expected Output:

W1	[[1.78862847 0.43650985] [0.09649747 -1.8634927] [-0.2773882 -0.35475898] [-0.08274148 -0.62700068]]
b1	[[0.] [0.] [0.] [0.]]
W2	[[-0.03098412 -0.33744411 -0.92904268 0.62552248]]
b2	[[0.]]

与预期结果一致；下面是迭代结果和分类效果：

```
Cost after iteration 5000: 0.5304944491717495
Cost after iteration 6000: 0.4138645817071795
Cost after iteration 7000: 0.3117803464844441
Cost after iteration 8000: 0.23696215330322565
Cost after iteration 9000: 0.18597287209206836
Cost after iteration 10000: 0.15015556280371817
Cost after iteration 11000: 0.12325079292273548
Cost after iteration 12000: 0.09917746546525937
Cost after iteration 13000: 0.0845705595402428
Cost after iteration 14000: 0.07357895962677366
```





效果很明显，正确率又从 83%提升到了 99%；

2. 对于梯度检验：

① 对于一维的梯度检验：

计算结果如图：

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

Expected Output:

The gradient is correct!

** difference **	2.9193358103083e-10

可以发现：计算检验正确。即，可以正确的计算反向传播；现在说明如果输入一个一维的数据是可以正确训练

的，但是一般 theta 是由 w 和 b 多维矩阵组成；

② 对于多维的结果检验：

```
X, Y, parameters = gradient_check_n_test_case()

cost, cache = forward_propagation_n(X, Y, parameters)
gradients = backward_propagation_n(X, Y, cache)
difference = gradient_check_n(parameters, gradients, X, Y)
```

```
There is a mistake in the backward propagation! difference = 0.285093156780699
```

Expected output:

** There is a mistake in the backward propagation!**	difference = 0.285093156781

经过该份代码原始的 forward prop 和 backward prop 的设计以及 gradient_check_n 的设计可以发现出现的结果和预期结果一致。

3. 优化：

① GD and SGD：

对于 GD 的结果展示：

```
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
W1 = [[ 1.63535156 -0.62320365 -0.53718766]
 [-1.07799357  0.85639907 -2.29470142]]
b1 = [[ 1.74604067]
 [-0.75184921]]
W2 = [[ 0.32171798 -0.25467393  1.46902454]
 [-2.05617317 -0.31554548 -0.3756023 ]
 [ 1.1404819  -1.09976462 -0.1612551 ]]
b2 = [[-0.88020257]
 [ 0.02561572]
 [ 0.57539477]]
```

Expected Output:

W1	[[1.63535156 -0.62320365 -0.53718766] [-1.07799357 0.85639907 -2.29470142]]
b1	[[1.74604067] [-0.75184921]]
W2	[[0.32171798 -0.25467393 1.46902454] [-2.05617317 -0.31554548 -0.3756023] [1.1404819 -1.09976462 -0.1612551]]
b2	[[-0.88020257] [0.02561572] [0.57539477]]

如图所示：输出结果和预期结果一致；

看一下随机梯度下降和上述的梯度下降的区别：

- **(Batch) Gradient Descent**:

```
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    # Forward propagation
    a, caches = forward_propagation(X, parameters)
    # Compute cost.
    cost = compute_cost(a, Y)
    # Backward propagation.
    grads = backward_propagation(a, caches, parameters)
    # Update parameters.
    parameters = update_parameters(parameters, grads)
```

- **Stochastic Gradient Descent**:

```
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    for j in range(0, m):
        # Forward propagation
        a, caches = forward_propagation(X[:,j], parameters)
        # Compute cost
        cost = compute_cost(a, Y[:,j])
        # Backward propagation
        grads = backward_propagation(a, caches, parameters)
        # Update parameters.
        parameters = update_parameters(parameters, grads)
```

在数据集特别大的时候，随机梯度下降收敛的更快；但是参数会朝着最小值不断的震荡，而不是平滑的收敛；再就是小批量梯度下降；而这三者梯度下降、小批量梯度下降和随机梯度下降之间的区别在于用于执行一个更新步骤的示例数量。必须调整学习率超参数。有了一个很好的小批量大小，它通常优于梯度下降或随机梯度下降（特别是当训练集很大时）。

② Mini-Batch Gradient descent：

结果展示：

```
shape of the 1st mini_batch_X: (12288, 64)
shape of the 2nd mini_batch_X: (12288, 64)
shape of the 3rd mini_batch_X: (12288,)
shape of the 1st mini_batch_Y: (1, 64)
shape of the 2nd mini_batch_Y: (1, 64)
shape of the 3rd mini_batch_Y: (1,)
mini batch sanity check: [ 0.90085595 -0.7612069   0.2344157 ]
```

Expected Output:

shape of the 1st mini_batch_X	(12288, 64)
shape of the 2nd mini_batch_X	(12288, 64)
shape of the 3rd mini_batch_X	(12288, 20)
shape of the 1st mini_batch_Y	(1, 64)
shape of the 2nd mini_batch_Y	(1, 64)
shape of the 3rd mini_batch_Y	(1, 20)
mini batch sanity check	[0.90085595 -0.7612069 0.2344157]

实验结果与预期结果一致；Shuffling 和 Partitioning 是构建小批量所需的两个步骤 ，并且通常选择 2 的幂作为小批量大小，例如 16、32、64、128。

③ Momentum：

初始化速度结果：

```
v["dW1"] = [[0. 0. 0.]
 [0. 0. 0.]]
v["db1"] = [[0.]
 [0.]]
v["dW2"] = [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
v["db2"] = [[0.]
 [0.]
 [0.]]
```

Expected Output:

v["dW1"]	[[0. 0. 0.] [0. 0. 0.]]
v["db1"]	[[0.] [0.]]
v["dW2"]	[[0. 0. 0.] [0. 0. 0.] [0. 0. 0.]]
v["db2"]	[[0.] [0.] [0.]]

参数更新结果：

```
W1 = [[ 1.62544598 -0.61290114 -0.52907334]
 [-1.07347112  0.86450677 -2.30085497]]
b1 = [[ 1.74493465]
 [-0.76027113]]
W2 = [[ 0.31930698 -0.24990073  1.4627996 ]
 [-2.05974396 -0.32173003 -0.38320915]
 [ 1.13444069 -1.0998786  -0.1713109 ]]
b2 = [[-0.87809283]
 [ 0.04055394]
 [ 0.58207317]]
```

Expected Output:

W1	[[1.62544598 -0.61290114 -0.52907334] [-1.07347112 0.86450677 -2.30085497]]
b1	[[1.74493465] [-0.76027113]]
W2	[[0.31930698 -0.24990073 1.4627996] [-2.05974396 -0.32173003 -0.38320915] [1.13444069 -1.0998786 -0.1713109]]
b2	[[-0.87809283] [0.04055394] [0.58207317]]
v["dW1"]	[[-0.11006192 0.11447237 0.09015907] [0.05024943 0.09008559 -0.06837279]]
v["db1"]	[[-0.01228902] [-0.09357694]]
v["dW2"]	[[-0.02678881 0.05303555 -0.06916608] [-0.03967535 -0.06871727 -0.08452056] [-0.06712461 -0.00126646 -0.11173103]]
**v["db2"]	[[0.02344157] [0.16598022] [0.07420442]]

结果和预期结果一致；Momentum 考虑过去的梯度以平滑梯度下降的步骤。 它可以应用于批量梯度下降、小批量梯度下降或随机梯度下降。调整动量超参数和学习率。

对于超参数 β 的选择：Momentum 越大，更新越平滑，因为我们越多地考虑过去的梯度。 但是如果太大，它也可能使更新过于平滑。范围从 0.8 到 0.999 的常用值。 为我们的模型调整最佳值可能需要尝试多个值，以查看在降低成本函数值方面最有效的值。

④ Adam:

结果展示：

```
W1 = [[ 1.63178673 -0.61919778 -0.53561312]
 [-1.08040999  0.85796626 -2.29409733]]
b1 = [[ 1.75225313]
 [-0.75376553]]
W2 = [[ 0.32648046 -0.25681174  1.46954931]
 [-2.05269934 -0.31497584 -0.37661299]
 [ 1.14121081 -1.09245036 -0.16498684]]
b2 = [[-0.88529978]
 [ 0.03477238]
 [ 0.57537385]]
```

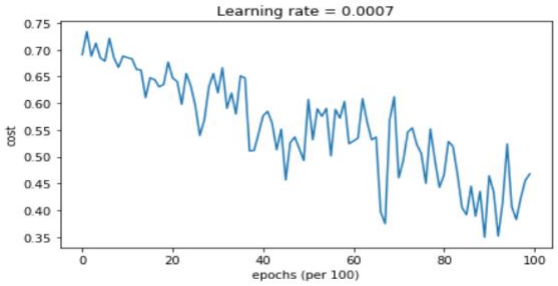
Expected Output:

W1	[[1.63178673 -0.61919778 -0.53561312] [-1.08040999 0.85796626 -2.29409733]]
b1	[[1.75225313] [-0.75376553]]
W2	[[0.32648046 -0.25681174 1.46954931] [-2.05269934 -0.31497584 -0.37661299] [1.14121081 -1.09245036 -0.16498684]]
b2	[[-0.88529978] [0.03477238] [0.57537385]]
v["dW1"]	[[-0.11006192 0.11447237 0.09015907] [0.05024943 0.09008559 -0.06837279]]
v["db1"]	[[-0.01228902] [-0.09357694]]
v["dW2"]	[[-0.02678881 0.05303555 -0.06916608] [-0.03967535 -0.06871727 -0.08452056] [-0.06712461 -0.00126646 -0.11173103]]

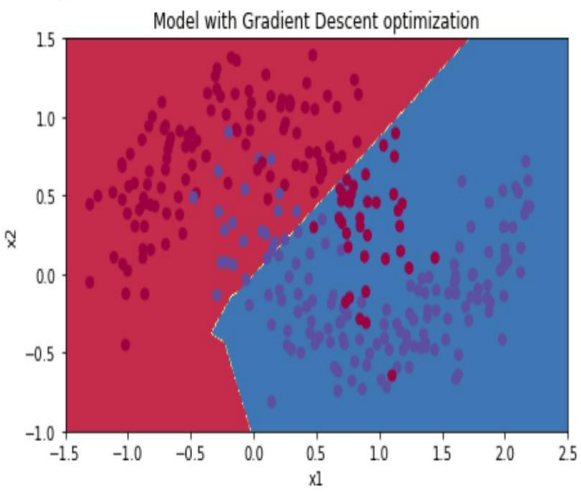
输出结果和预期结果一致；

⑤ Model:

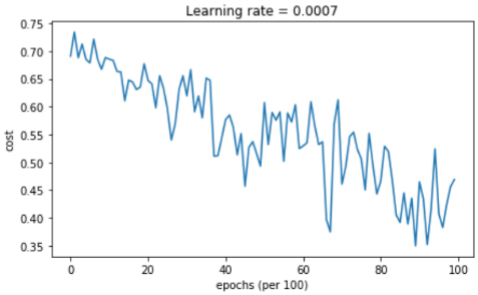
```
Cost after epoch 0: 0.690736
Cost after epoch 1000: 0.685273
Cost after epoch 2000: 0.647072
Cost after epoch 3000: 0.619525
Cost after epoch 4000: 0.576584
Cost after epoch 5000: 0.607243
Cost after epoch 6000: 0.529403
Cost after epoch 7000: 0.460768
Cost after epoch 8000: 0.465586
Cost after epoch 9000: 0.464518
```



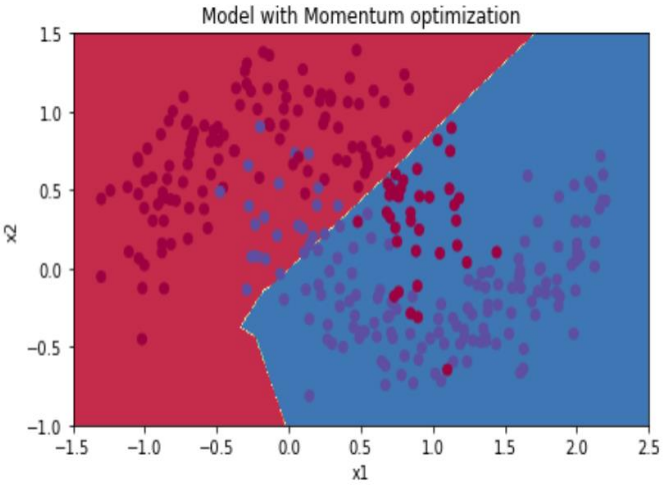

```
Accuracy: 0.796666666667
```



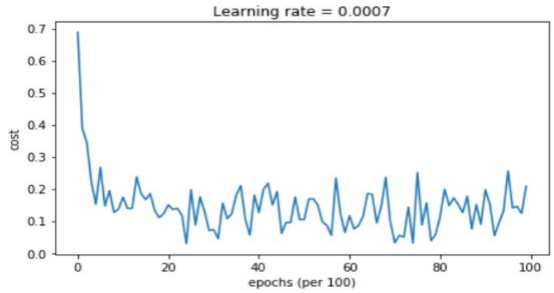

```
Cost after epoch 0: 0.690741
Cost after epoch 1000: 0.685341
Cost after epoch 2000: 0.647145
Cost after epoch 3000: 0.619594
Cost after epoch 4000: 0.576665
Cost after epoch 5000: 0.607324
Cost after epoch 6000: 0.529476
Cost after epoch 7000: 0.460936
Cost after epoch 8000: 0.465780
Cost after epoch 9000: 0.464740
```



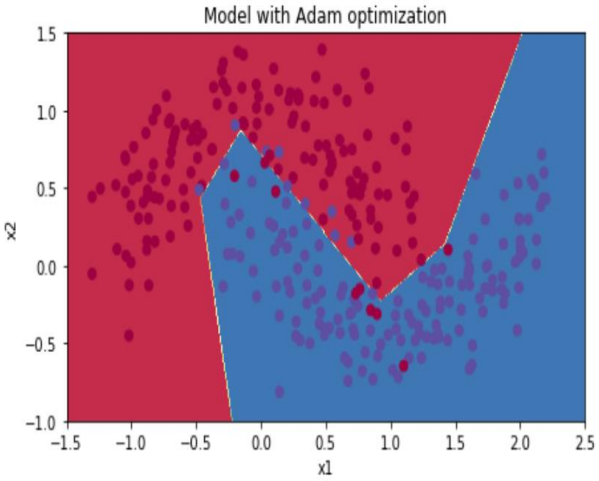

```
Accuracy: 0.796666666667
```




```
Cost after epoch 0: 0.687550
Cost after epoch 1000: 0.173593
Cost after epoch 2000: 0.150145
Cost after epoch 3000: 0.072939
Cost after epoch 4000: 0.125896
Cost after epoch 5000: 0.104185
Cost after epoch 6000: 0.116069
Cost after epoch 7000: 0.031774
Cost after epoch 8000: 0.112908
Cost after epoch 9000: 0.197732
```




```
Accuracy: 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. 对于多维的梯度检验，我们检查出来了 backward prop 的错误进行修复：
   根据提示修改了 dw2 和 db1：但是出现了以下情况：

```
There is a mistake in the backward propagation! difference = 1.1890417878779317e-07
```

仍然显示错误但是后面的 difference 确实出现了变化；虽然没得到一个完美结果，但是相比之前有所提升。