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：Homework2		学号：201600122057
日期：2019.3.18	班级： 16 智能	姓名： 贝仕成
Email：337263318@qq.com		

实验目的：
用提取的图片特征取代图片的像素值作为神经网络的输入，观察实验结果。
同时学习一系列改进模型的方法（gradient checking，initialization，optimization methods，regularization），并用这些方法提升实验效果。

实验软件和硬件环境：
Python 3.5.6
Jupyter notebook 5.0.0

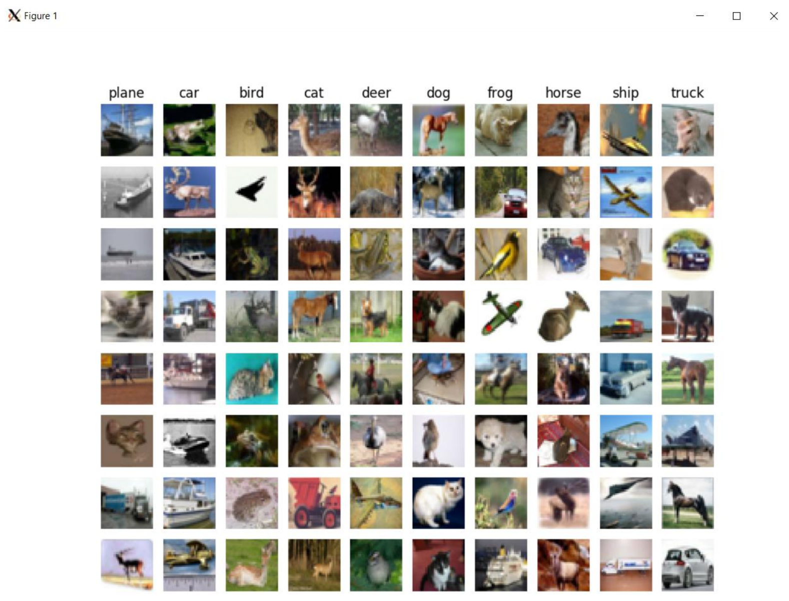
神舟战神 Z7M-KP7S1
NVIDIA GTX1050Ti

实验原理和方法：
梯度检查，参数初始化，梯度下降及相关优化方法，正则化技术

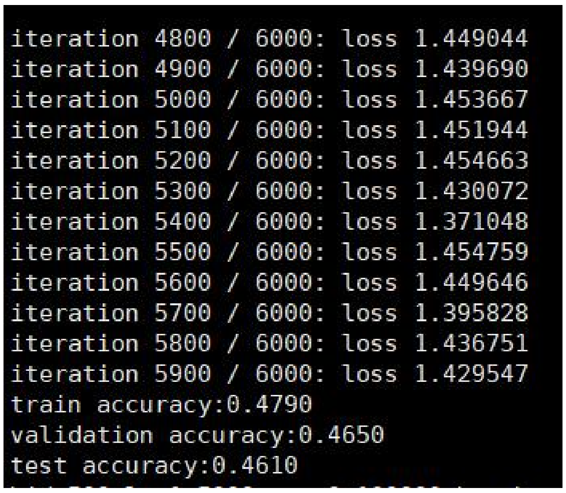
实验步骤：（不要求罗列完整源代码）：

1. features.py：

错误可视化：

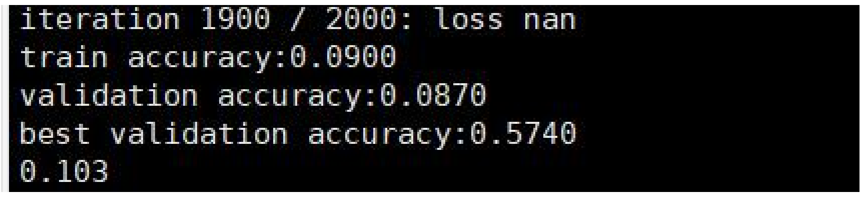


老实说，参数很难调，试了很多次 loss 都是在 2.3 左右徘徊，就是降不下去，在尝试将学习率拔高到 1.8，batch_size 拔高到 2000 后，出现了让人能看的结果。



```
iteration 4800 / 6000: loss 1.449044
iteration 4900 / 6000: loss 1.439690
iteration 5000 / 6000: loss 1.453667
iteration 5100 / 6000: loss 1.451944
iteration 5200 / 6000: loss 1.454663
iteration 5300 / 6000: loss 1.430072
iteration 5400 / 6000: loss 1.371048
iteration 5500 / 6000: loss 1.454759
iteration 5600 / 6000: loss 1.449646
iteration 5700 / 6000: loss 1.395828
iteration 5800 / 6000: loss 1.436751
iteration 5900 / 6000: loss 1.429547
train accuracy:0.4790
validation accuracy:0.4650
test accuracy:0.4610
```

但最终结果却是：

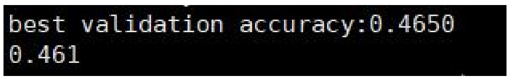


```
iteration 1900 / 2000: loss nan
train accuracy:0.0900
validation accuracy:0.0870
best validation accuracy:0.5740
0.103
```

我很费解，为什么储存下来的最好的模型对测试集的效果这么差，只有 0.103。检查了几遍发现 best_net 实际上保存的是最后一次训练的网络，因为我缺少了一行比较关键的代码：

```
net = ThreeLayerNet(input_dim, hidden_dim, num_classes)
```

这一行少了的话，best_net=net 的存在导致每一次 net.train 后 best_net 也会跟着发生改变。改变了以后，结果变好了点。但相比于上次的实验作业，这个结果并不能让人满意。



```
best validation accuracy:0.4650
0.461
```

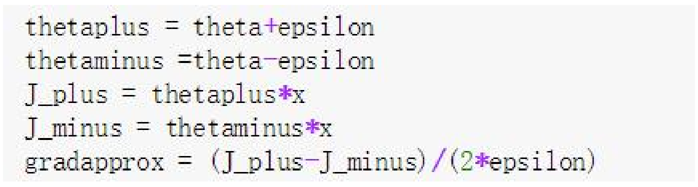
暴力调参有很耗时，还有很大可能出不来结果。所以必须考虑一些优化模型的方法来改进模型。

2. Optimization

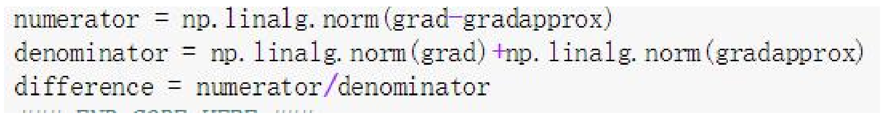
2.1 Gradient checking

主要是用来计算反向传播时梯度计算是否正确，与其说优化它应是一个检验工具。

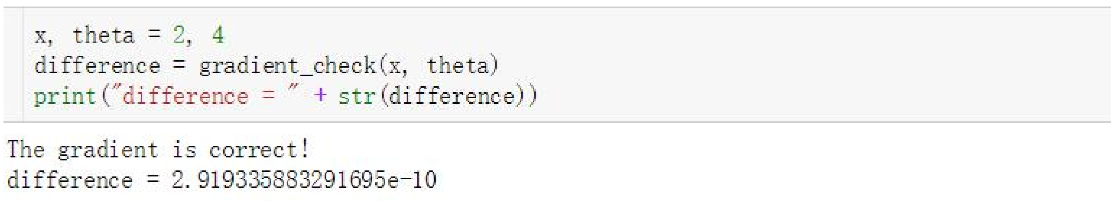
以下贴出 notebook 上的一些重要的结果：



```
thetaplus = theta+epsilon
thetaminus =theta-epsilon
J_plus = thetaplus*x
J_minus = thetaminus*x
gradapprox = (J_plus-J_minus)/(2*epsilon)
```

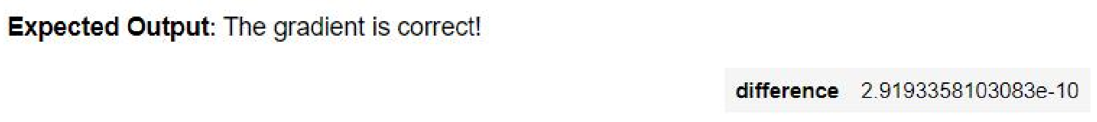


```
numerator = np.linalg.norm(grad-gradapprox)
denominator = np.linalg.norm(grad)+np.linalg.norm(gradapprox)
difference = numerator/denominator
```



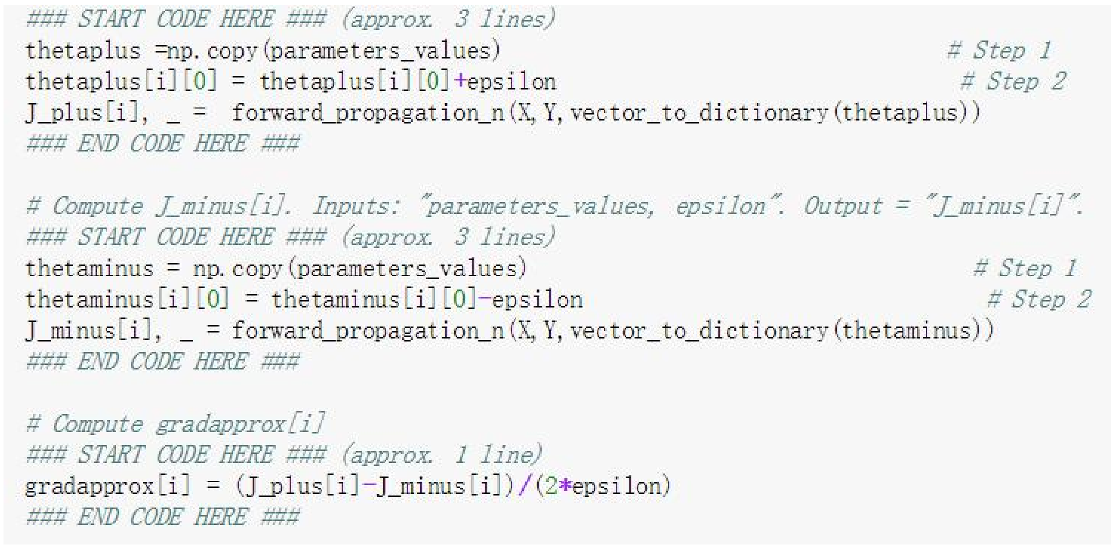
```
x, theta = 2, 4
difference = gradient_check(x, theta)
print("difference = " + str(difference))
```

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

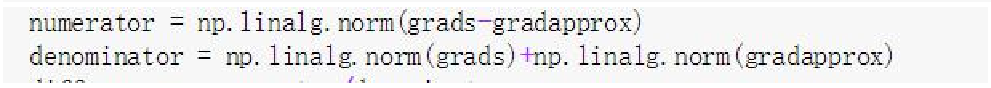


Expected Output: The gradient is correct!

difference	2.9193358103083e-10

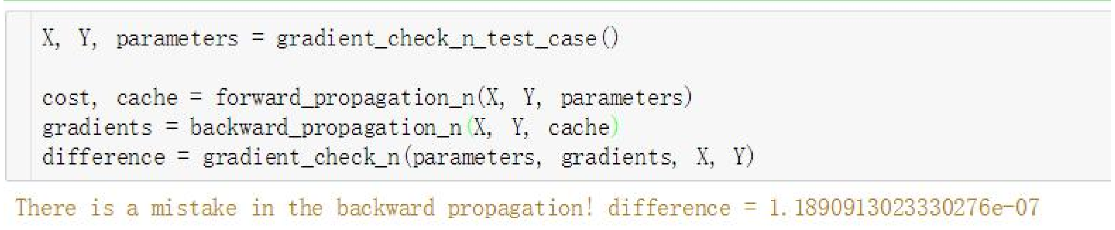


```
### START CODE HERE ### (approx. 3 lines)
thetaplus =np.copy(parameters_values)                                  # Step 1
thetaplus[i][0] = thetaplus[i][0]+epsilon                               # Step 2
J_plus[i], _ =  forward_propagation_n(X,Y,vector_to_dictionary(thetaplus))
### END CODE HERE ###

# 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))
### 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 ###
```



```
numerator = np.linalg.norm(grads-gradapprox)
denominator = np.linalg.norm(grads)+np.linalg.norm(gradapprox)
```

按要求改正了一些代码后结果还是没有低于 1e-7：



```
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)
```

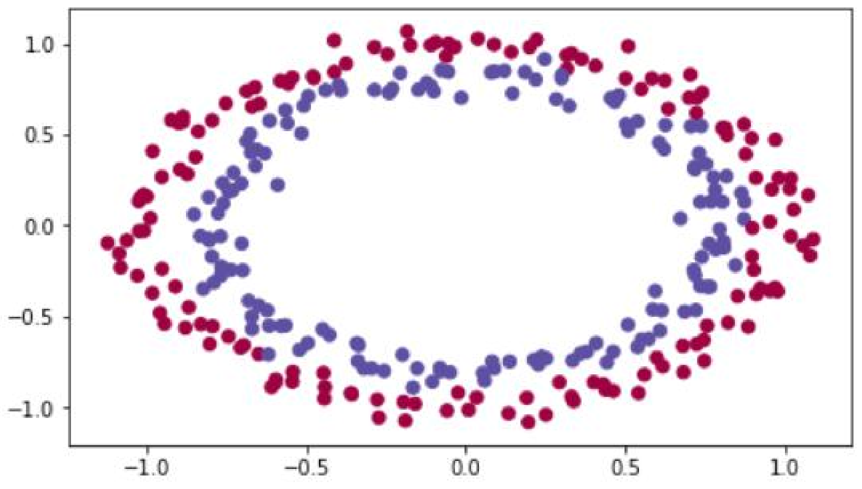
```
There is a mistake in the backward propagation! difference = 1.1890913023330276e-07
```

Gradient checking 证实了反向传播时梯度的计算值和梯度的数值估计之间的相近。但是这个过程比较慢，所以不必每个迭代都用上它。当然如果你足够自信认为自己算的绝对没错，

那就用不上它。

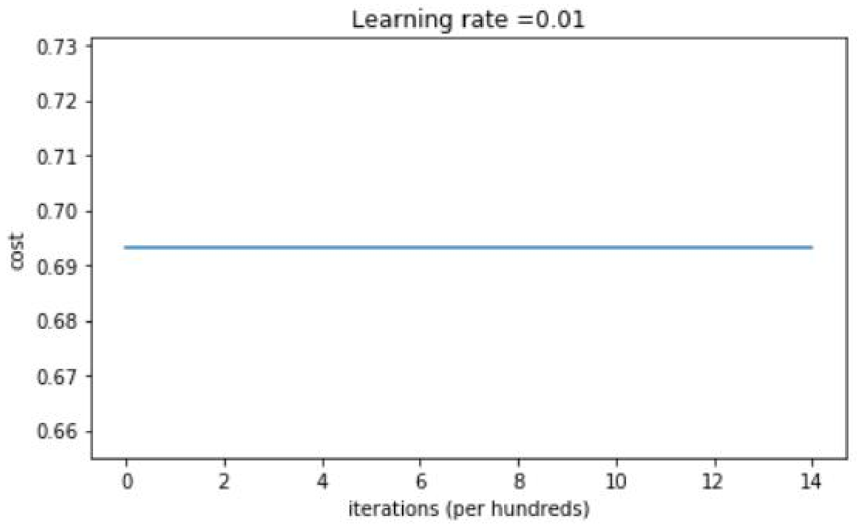
2.2 Initialization

初始化各个参数矩阵，除了一般的随机初始化以外，还有一些其他的方法。

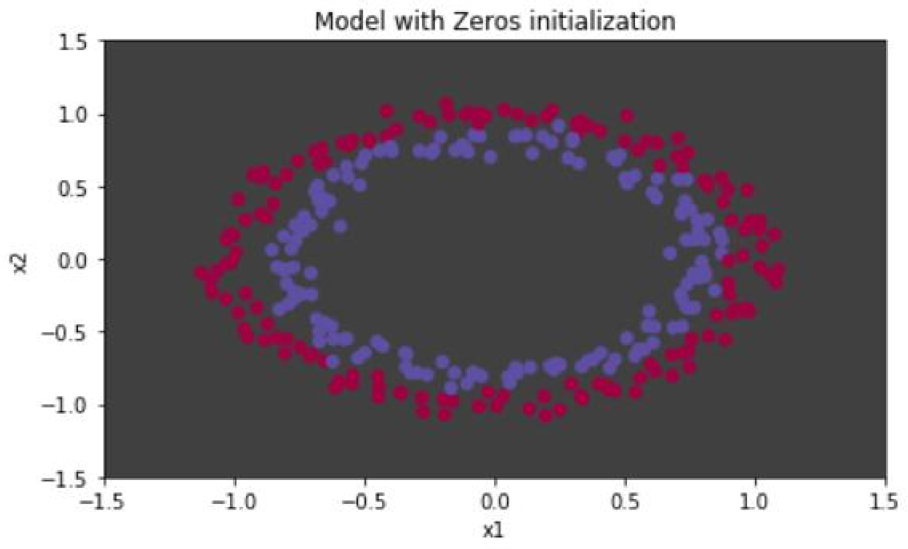


模型需要训练得出一个可以区分红蓝数据点的边界。

①Zero initialization



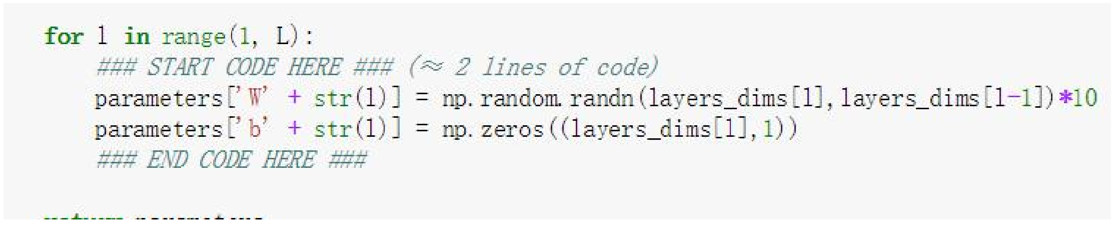
```
On the train set:
Accuracy: 0.5
On the test set:
Accuracy: 0.5
```



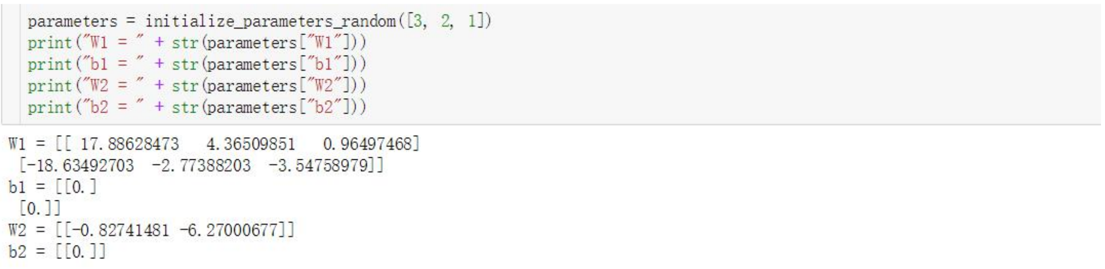
对每个样本模型都预测为 0，这样预测对于这样一个红蓝数据点个数相等的数据集来说准确率当然只有 50%，最后显示的图也没有任何分界线的出现。

②Random initialization

随机初始化参数矩阵。

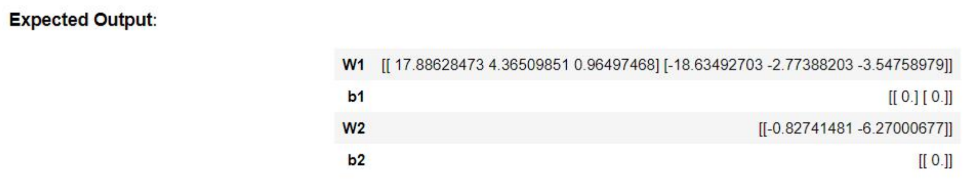


```
for l in range(1, L):
    ### START CODE HERE ### (≈ 2 lines of code)
    parameters['W' + str(l)] = np.random.randn(layers_dims[l],layers_dims[l-1])*10
    parameters['b' + str(l)] = np.zeros((layers_dims[l],1))
    ### END CODE HERE ###
```



```
parameters = initialize_parameters_random([3, 2, 1])
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

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

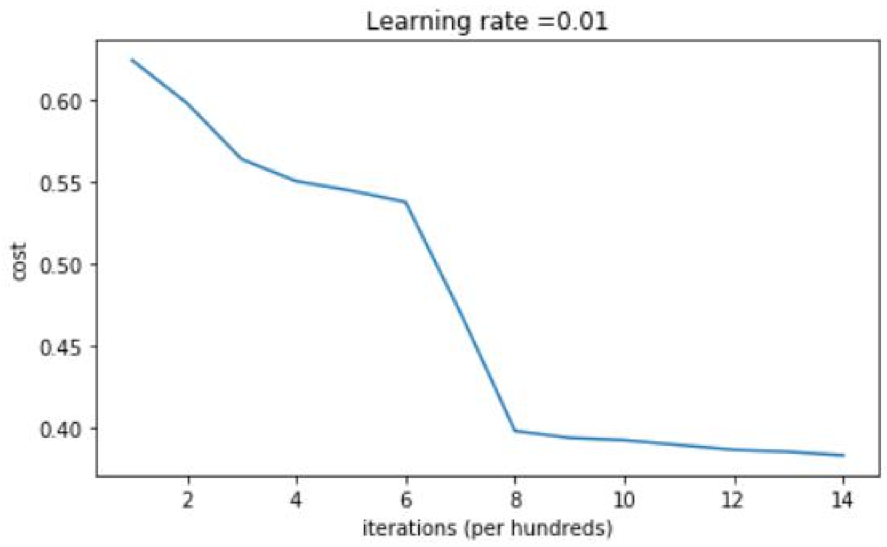


Expected Output:

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

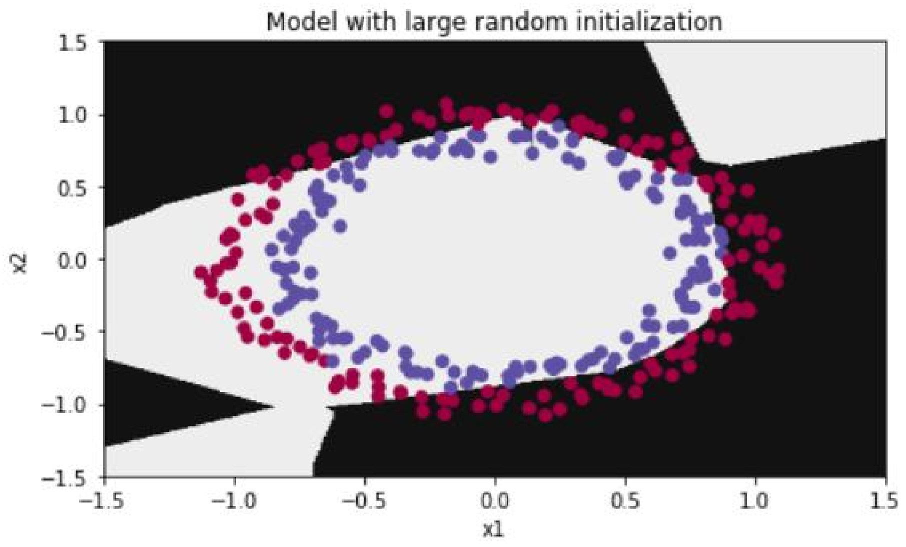
结果是一致的。

训练模型的结果如下：



```
On the train set:
Accuracy: 0.83
On the test set:
Accuracy: 0.86
```

相对于前种初始化方法 loss 有明显的下降，准确率有明显提升。

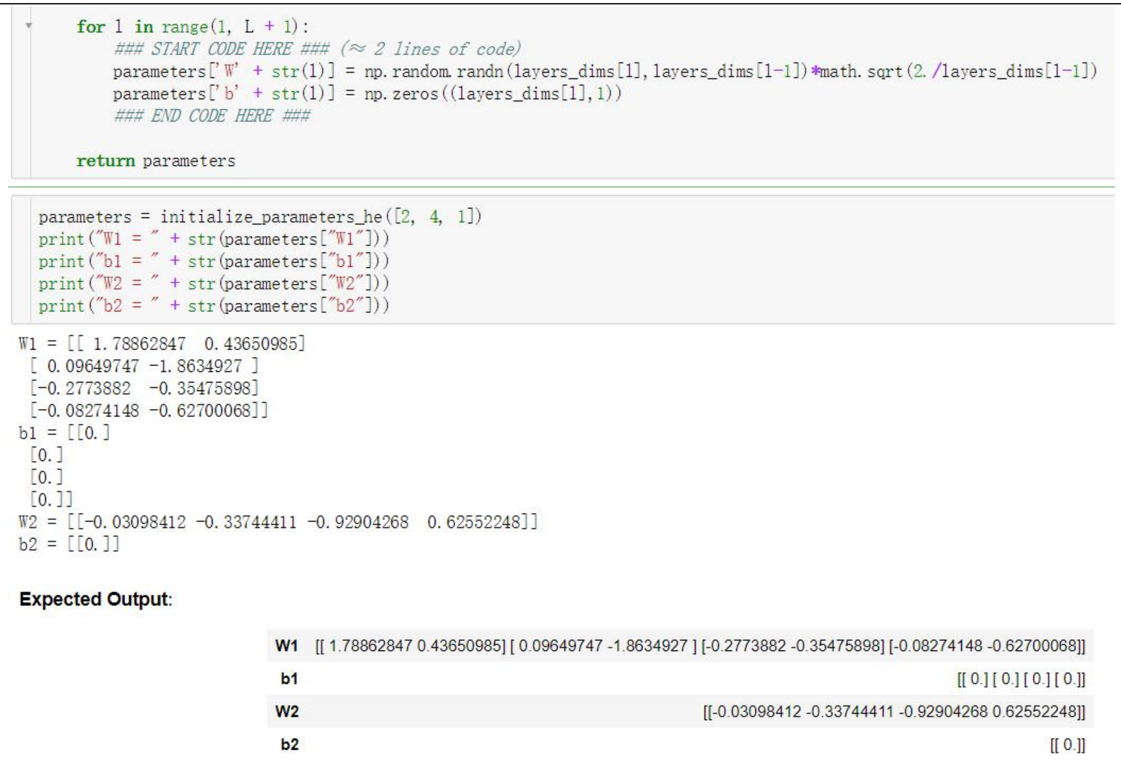


可以看到区域边界的划分，尽管这划分并算不上好。

np.random.randn(..,...)*10 初始化后的训练模型效果没有不乘十的效果好，notebook 上也说明随机初始化过大的初始值并不会产生很好的效果。

③He initialization

相对于上一种初始化方法 np.random.randn(..,...)*10，此方法以*math.sqrt(2./layers_dims[l-1])取代*10。



```
    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])*math.sqrt(2./layers_dims[l-1])
        parameters['b' + str(l)] = np.zeros((layers_dims[l],1))
        ### END CODE HERE ###

    return parameters
```

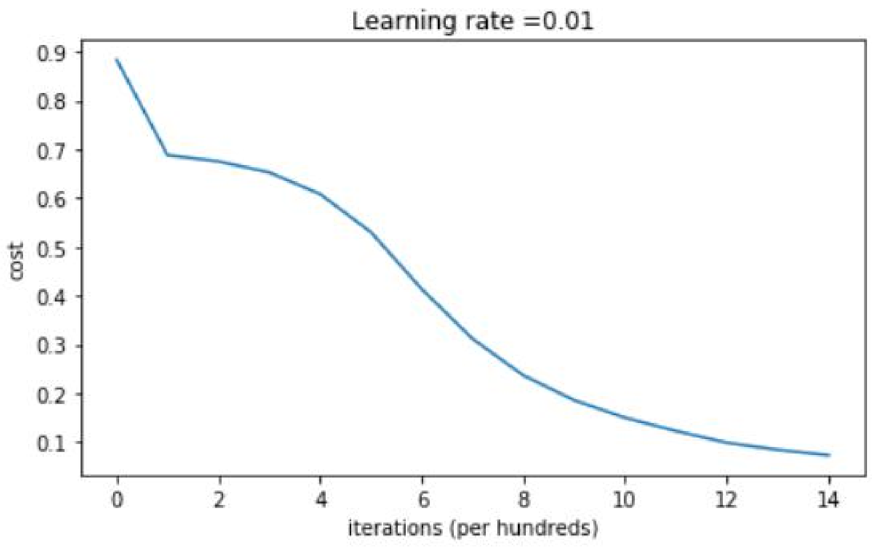
```
parameters = initialize_parameters_he([2, 4, 1])
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
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.]]
```

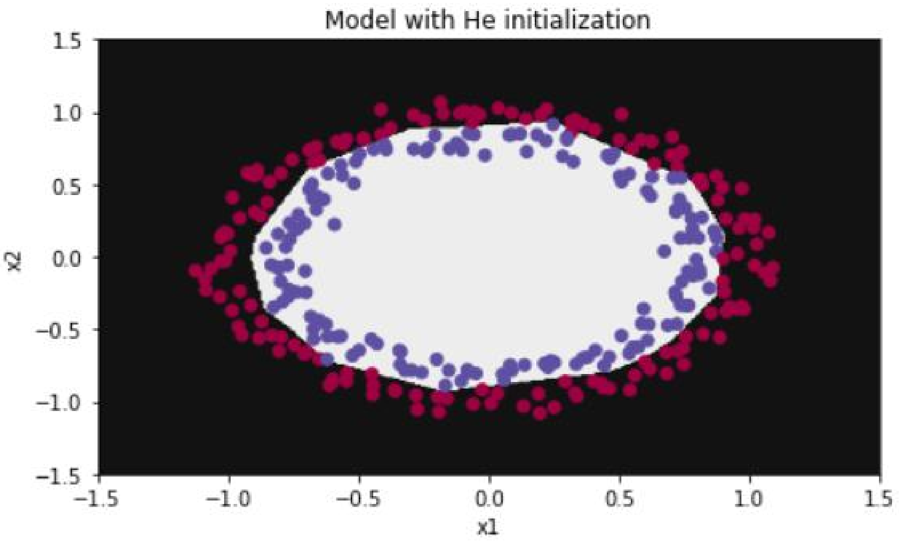
Expected Output:

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.]]

实验结果如下：



```
On the train set:
Accuracy: 0.9933333333333333
On the test set:
Accuracy: 0.96
```



可以看出 loss 曲线平滑了许多，模型对数据集的准确率得到了进一步的提升，图示的区域分界也基本准确。

Model	Train accuracy	Problem/Comment
3-layer NN with zeros initialization	50%	fails to break symmetry
3-layer NN with large random initialization	83%	too large weights
3-layer NN with He initialization	99%	recommended method

不同的初始化方法会导向不同的结果，random initialization 的 random 出来的值不应该过大，目前看来 He initialization 的效果是最好的，可以考虑用来改进上面输入图片特征的三层网络的模型。

2.3 Optimization methods

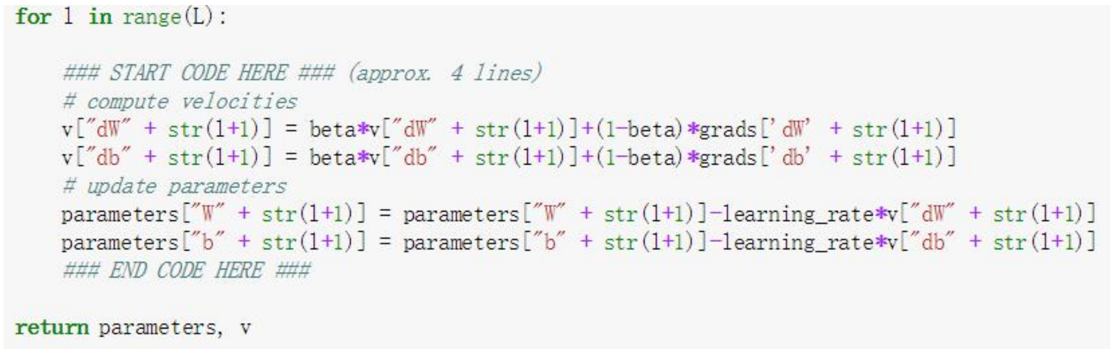
①Gradient descent

梯度下降分为一般梯度下降，随机梯度下降，mini-batch 梯度下降，带动量的梯度下降。其中带动量的梯度下降比较重要，它可以与前三种方法结合，贴出它的关键部分代码：

implement the parameters update with momentum. The momentum update rule is, for $l = 1, \dots, L$:

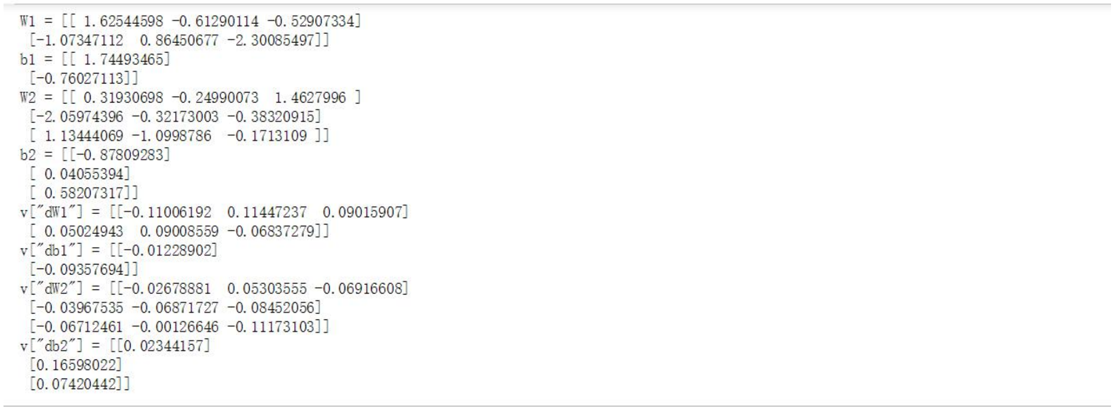
$$\begin{cases} v_{dW^{[l]}} = \beta v_{dW^{[l]}} + (1-\beta) dW^{[l]} \\ W^{[l]} = W^{[l]} - \alpha v_{dW^{[l]}} \end{cases}$$

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

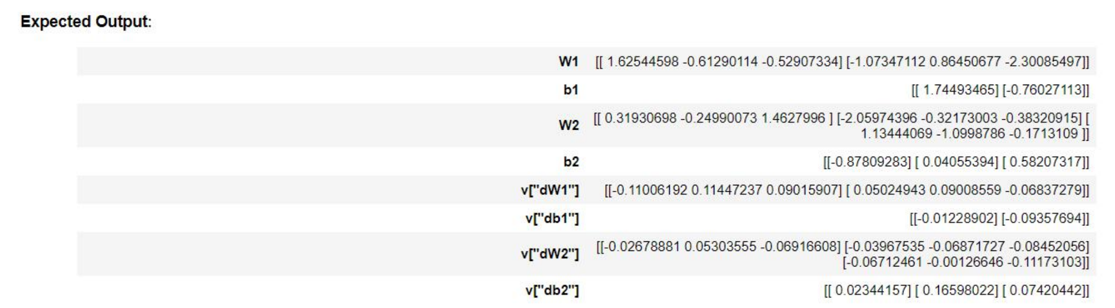


```
for l in range(L):

    ### 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 ###

return parameters, v
```

测试结果：



```
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]]
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]]
```

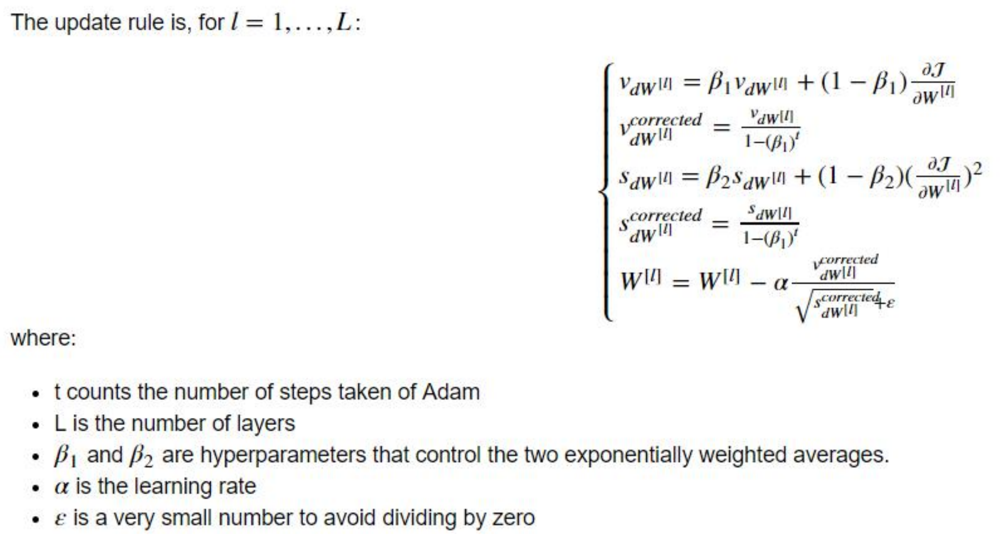


Expected Output:

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]]

β的值越大，本次梯度下降时的梯度会考虑更多上一次梯度的成分，β为0的话，这就成了一般的梯度下降。β一般取0.8-0.999，一般0.9为默认值。

②Adam

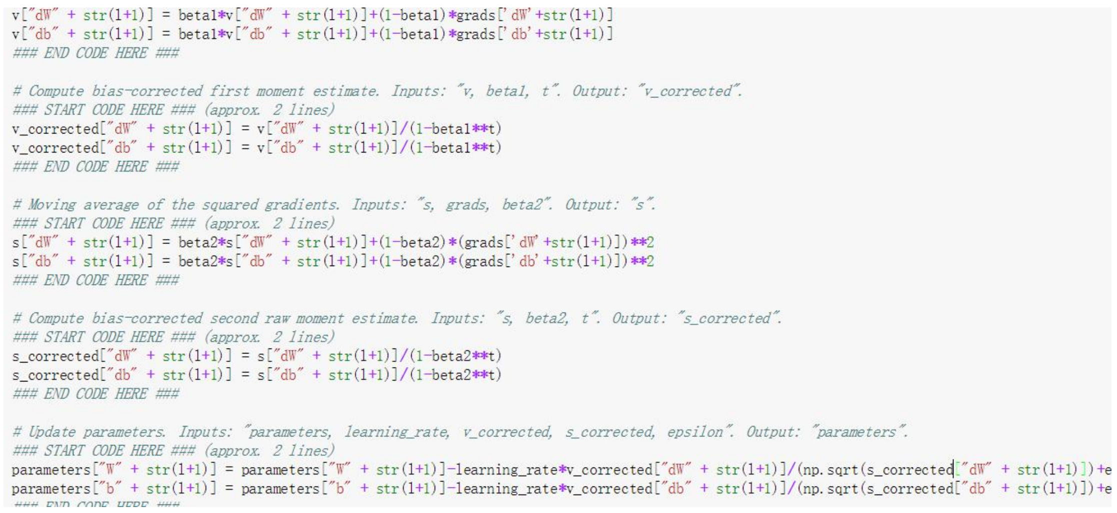

The update rule is, for $l = 1, \ldots, L$:

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

where:

- t counts the number of steps taken of Adam
- L is the number of layers
- β_1 and β_2 are hyperparameters that control the two exponentially weighted averages.
- α is the learning rate
- ε is a very small number to avoid dividing by zero

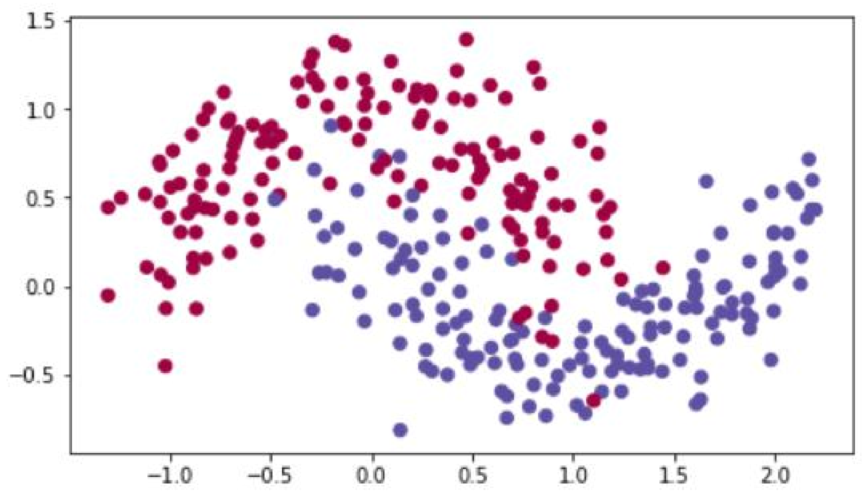
关键代码如下：



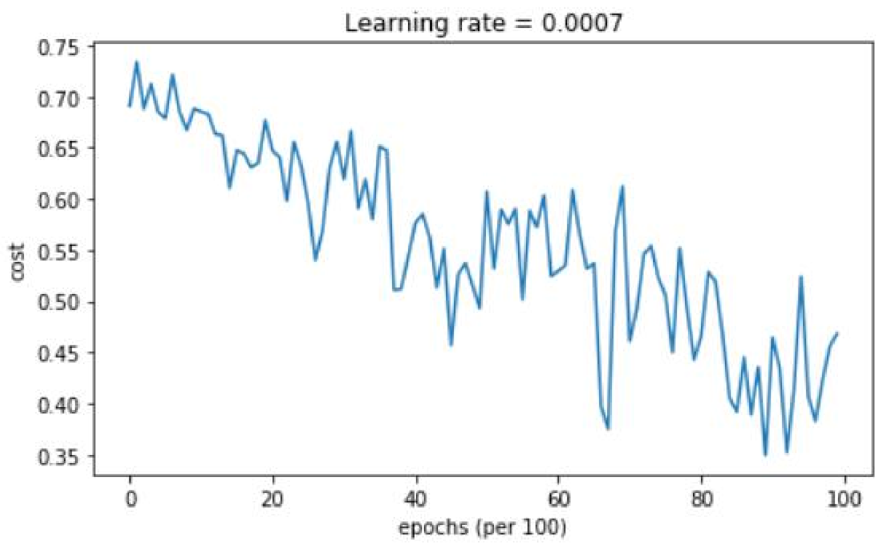
```
v["dW" + str(l+1)] = beta1*v["dW" + str(l+1)]+(1-beta1)*grads['dW'+str(l+1)]
v["db" + str(l+1)] = beta1*v["db" + str(l+1)]+(1-beta1)*grads['db'+str(l+1)]
### END CODE HERE ###

# 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-beta2**t)
s_corrected["db" + str(l+1)] = s["db" + str(l+1)]/(1-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)])+e
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)])+e
### END CODE HERE ###
```

下面对模型用各种优化方法进行实验：

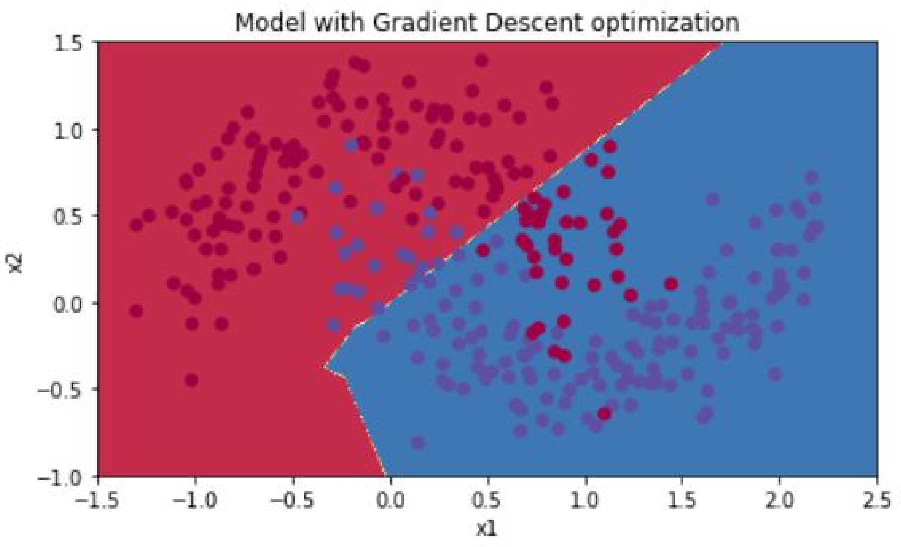
数据集的数据点图如下：



mini-batch gradient descent

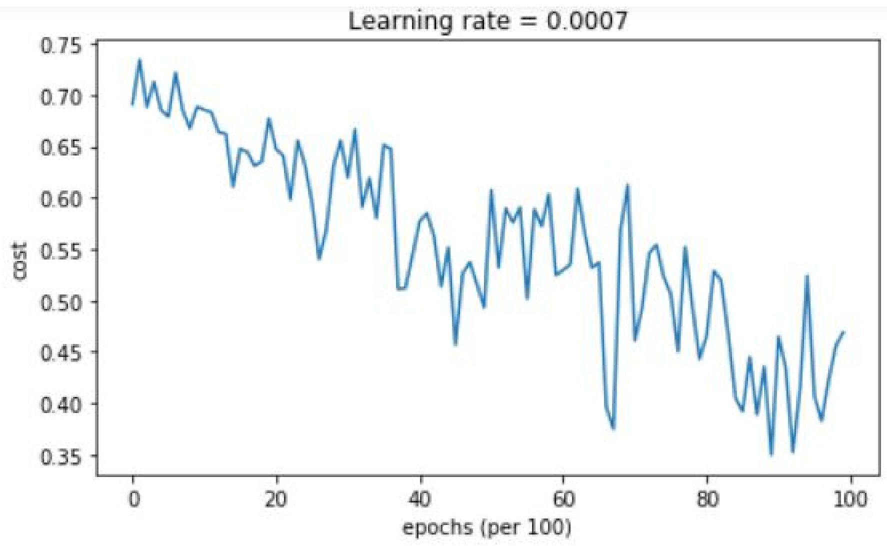


```
Accuracy: 0.7966666666666666
```

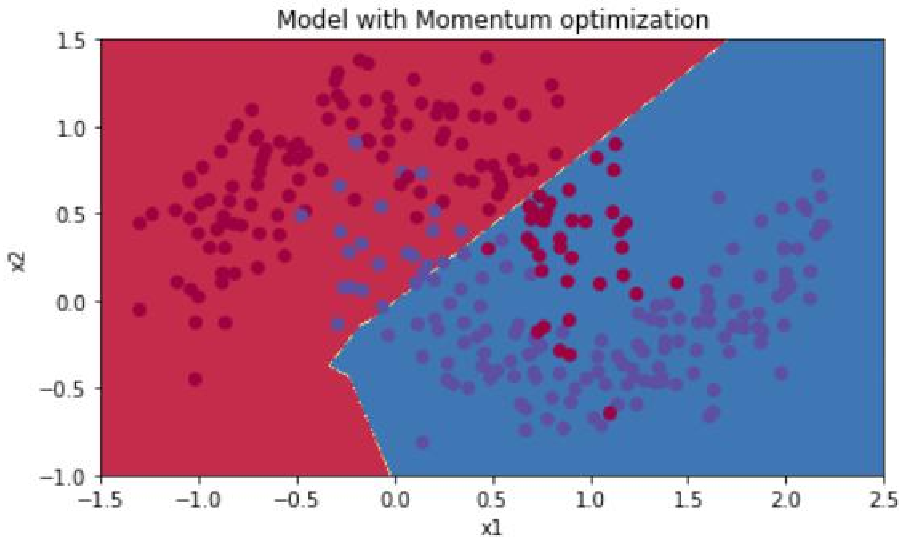


准确率还说的过去，但分界线着实不漂亮。

mini-batch gradient descent with momentum

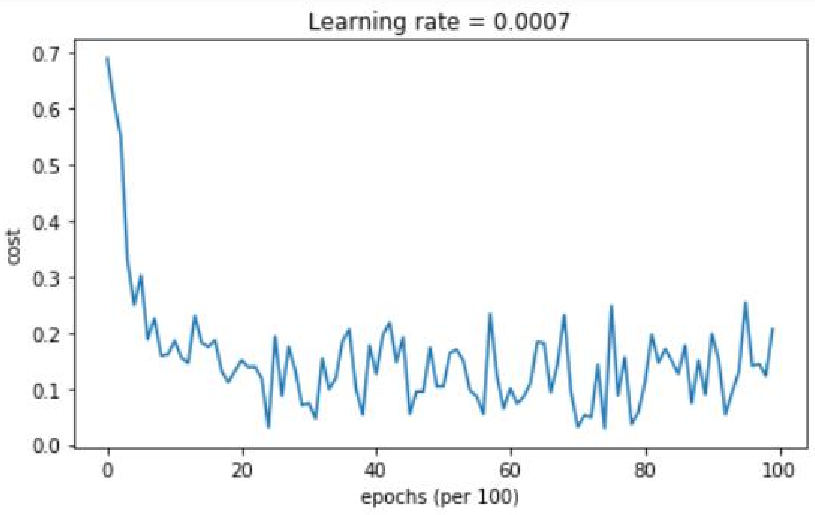


```
Accuracy: 0.7966666666666666
```

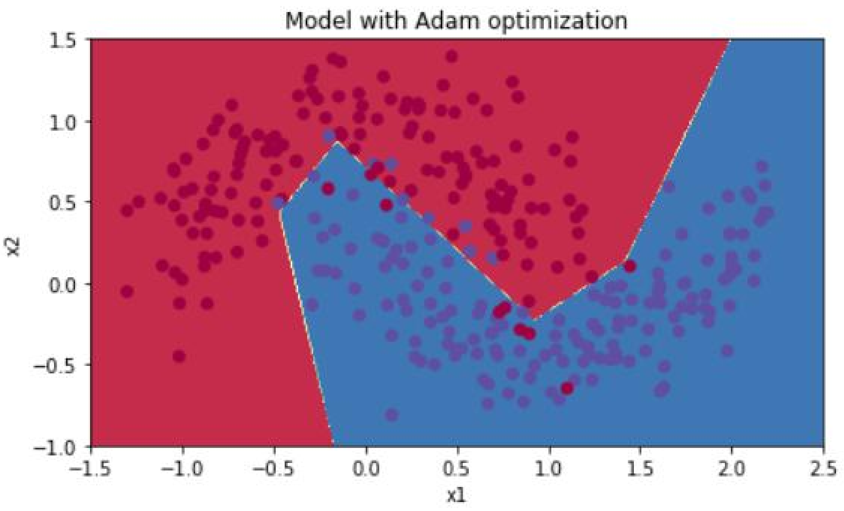


和不带动量的梯度下降表现并无什么区别

Mini-batch gradient descent with Adam mode



```
Accuracy: 0.94
```



结果提升非常明显，分界线也较好地符合了两类数据点的分布情况。

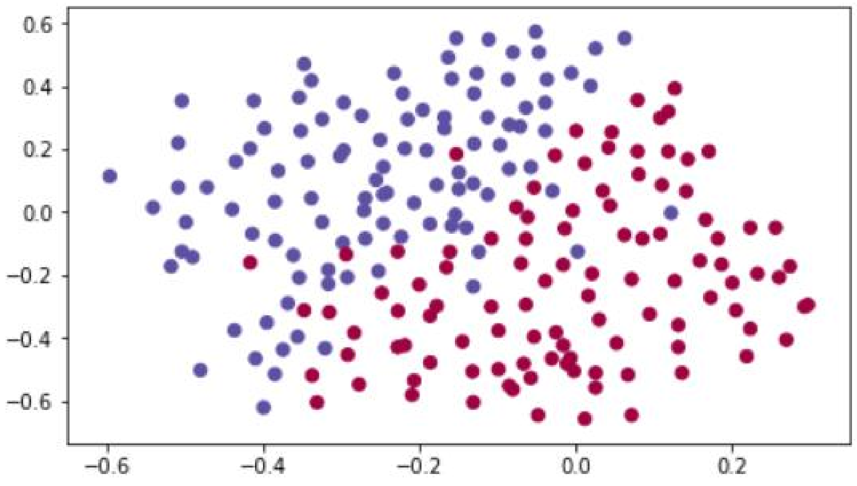
带动量的梯度下降通常是有不错效果的，但是对于过小的学习率或是过于简单化的数据集，它所带来的效果提升是可以忽视的。Adam 的表现优于上述其他方法，它很适合用于超参的调整过程。

2.4 Regularization

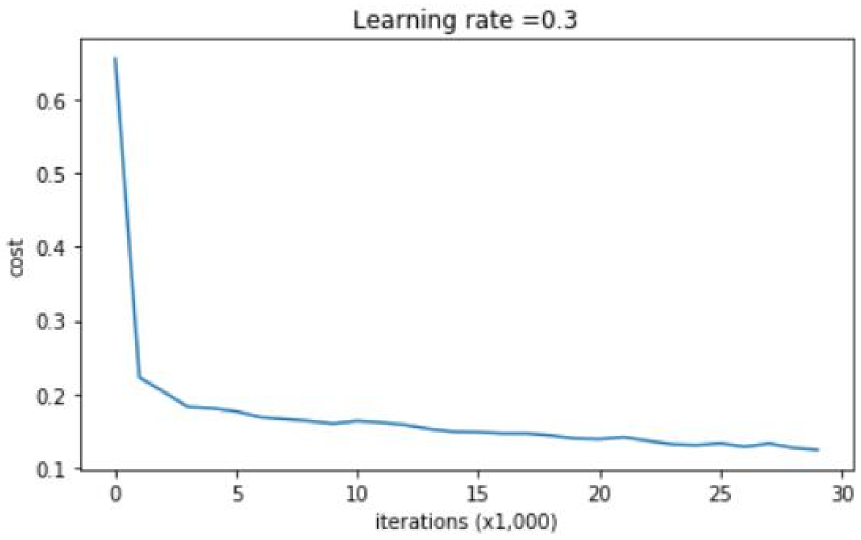
正则化技术其实之前的三层网络的模型就已经用到了。

载入数据集的数据点图：

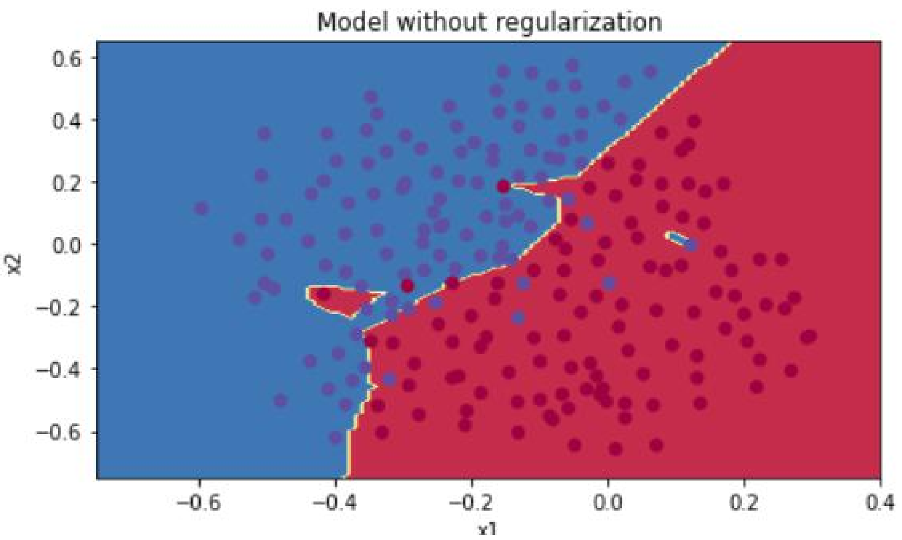
```
train_X, train_Y, test_X, test_Y = load_2D_dataset()
```



①Non-regularized model:



```
On the training set:
Accuracy: 0.9478672985781991
On the test set:
Accuracy: 0.915
```

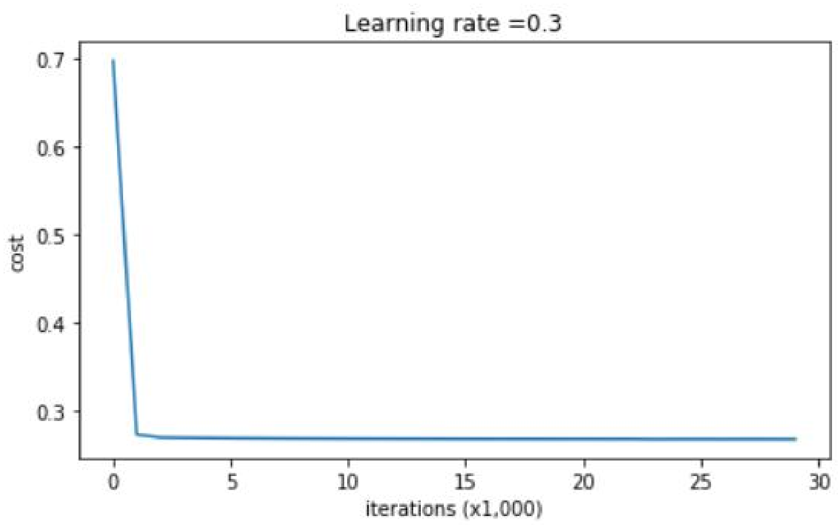


从上面的分布划分可以看出，该模型存在一点过拟合的状况。

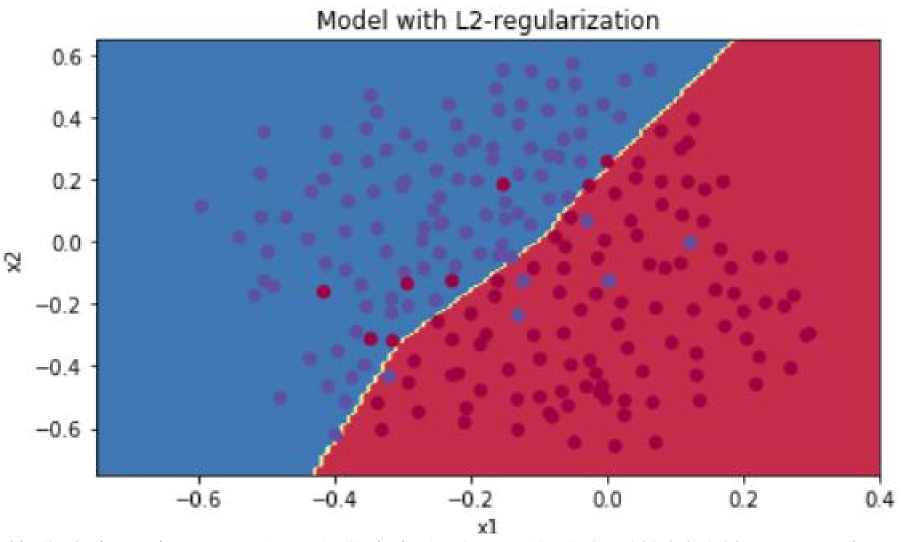
②L2 Regularization:

$$J = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log\left(a^{[L](i)}\right) + (1 - y^{(i)}) \log\left(1 - a^{[L](i)}\right) \right)$$

$$J_{regularized} = \underbrace{-\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log\left(a^{[L](i)}\right) + (1 - y^{(i)}) \log\left(1 - a^{[L](i)}\right) \right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m} \frac{\lambda}{2} \sum_l \sum_k \sum_j W_{k,j}^{[l]2}}_{\text{L2 regularization cost}}$$

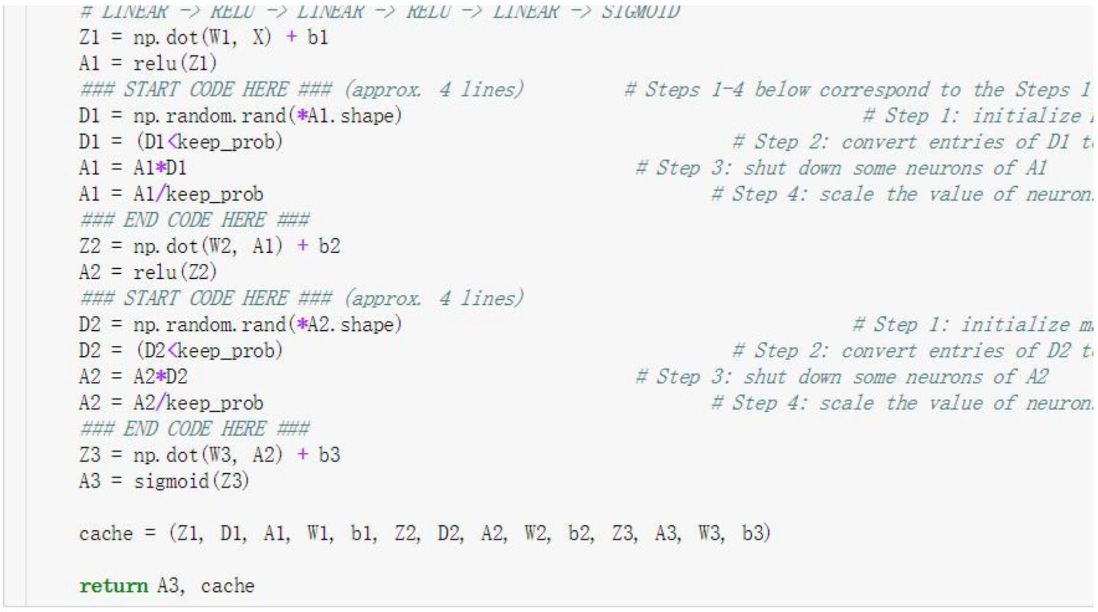


```
On the train set:
Accuracy: 0.9383886255924171
On the test set:
Accuracy: 0.93
```

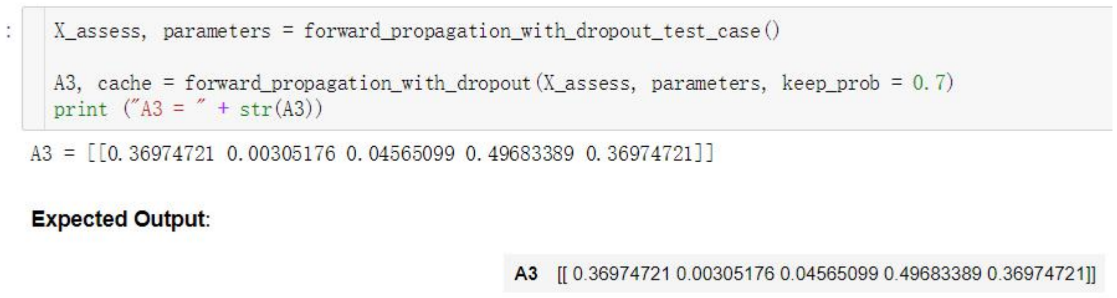


虽然训练准确率下降了，但是测试准确率上升了，从分布图的划分情况可以看出没有过拟合的情况出现。

③Dropout
dropout 技术会在每次迭代中随机关闭一些神经元，它也可以缓解过拟合情况的发生。Dropout 如果要使用的话，前向传播和反向传播都需要进行一定的操作。



```
    # LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID
    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)
    ### START CODE HERE ### (approx. 4 lines)         # Steps 1-4 below correspond to the Steps 1
    D1 = np.random.rand(*A1.shape)                                         # Step 1: initialize
    D1 = (D1<keep_prob)                                        # Step 2: convert entries of D1 t
    A1 = A1*D1                                         # Step 3: shut down some neurons of A1
    A1 = A1/keep_prob                                          # Step 4: scale the value of neuron
    ### END CODE HERE ###
    Z2 = np.dot(W2, A1) + b2
    A2 = relu(Z2)
    ### START CODE HERE ### (approx. 4 lines)
    D2 = np.random.rand(*A2.shape)                                         # Step 1: initialize m
    D2 = (D2<keep_prob)                                        # Step 2: convert entries of D2 t
    A2 = A2*D2                                         # Step 3: shut down some neurons of A2
    A2 = A2/keep_prob                                          # Step 4: scale the value of neuron
    ### END CODE HERE ###
    Z3 = np.dot(W3, A2) + b3
    A3 = sigmoid(Z3)

    cache = (Z1, D1, A1, W1, b1, Z2, D2, A2, W2, b2, Z3, A3, W3, b3)

    return A3, cache
```



```
:  X_assess, parameters = forward_propagation_with_dropout_test_case()

   A3, cache = forward_propagation_with_dropout(X_assess, parameters, keep_prob = 0.7)
   print ("A3 = " + str(A3))
```

```
A3 = [[0.36974721 0.00305176 0.04565099 0.49683389 0.36974721]]
```

Expected Output:

A3	[[0.36974721 0.00305176 0.04565099 0.49683389 0.36974721]]

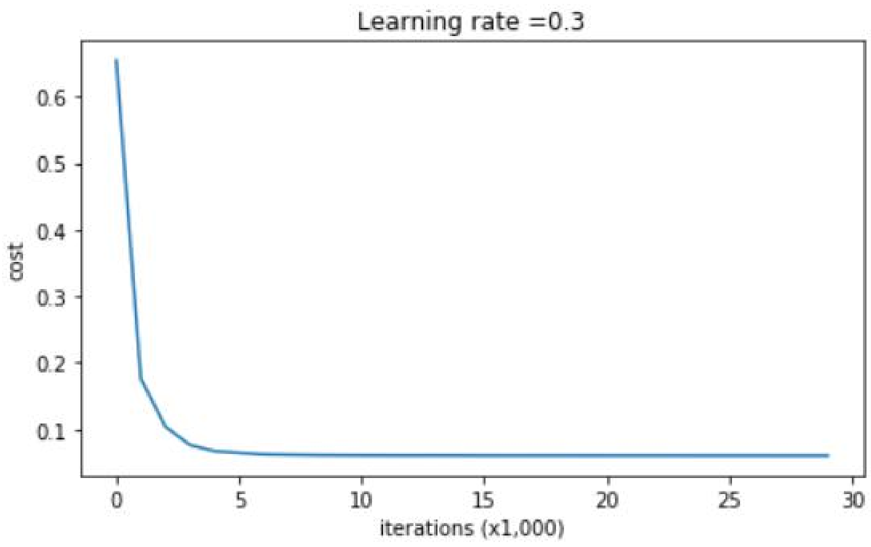
```
dZ3 = A3 - Y
dW3 = 1./m * np.dot(dZ3, A2.T)
db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)
dA2 = np.dot(W3.T, dZ3)
### START CODE HERE ### (≈ 2 lines of code)
dA2 = dA2*D2              # Step 1: Apply mask D2 to shut down the same neurons as during the forward propagation
dA2 = dA2/keep_prob            # Step 2: Scale the value of neurons that haven't been shut down
### END CODE HERE ###
dZ2 = np.multiply(dA2, np.int64(A2 > 0))
dW2 = 1./m * np.dot(dZ2, A1.T)
db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)

dA1 = np.dot(W2.T, dZ2)
### START CODE HERE ### (≈ 2 lines of code)
dA1 = dA1*D1              # Step 1: Apply mask D1 to shut down the same neurons as during the forward propagation
dA1 = dA1/keep_prob             # Step 2: Scale the value of neurons that haven't been shut down
### END CODE HERE ###
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)

gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3,"dA2": dA2,
             "dZ2": dZ2, "dW2": dW2, "db2": db2, "dA1": dA1,
             "dZ1": dZ1, "dW1": dW1, "db1": db1}
```

```
dA1 = [[ 0.36544439  0.         -0.00188233  0.         -0.17408748]
 [ 0.65515713  0.         -0.00337459  0.         -0.        ]]
dA2 = [[ 0.58180856  0.         -0.00299679  0.         -0.27715731]
 [ 0.          0.53159854 -0.          0.53159854 -0.34089673]
 [ 0.          0.         -0.00292733  0.         -0.        ]]
```

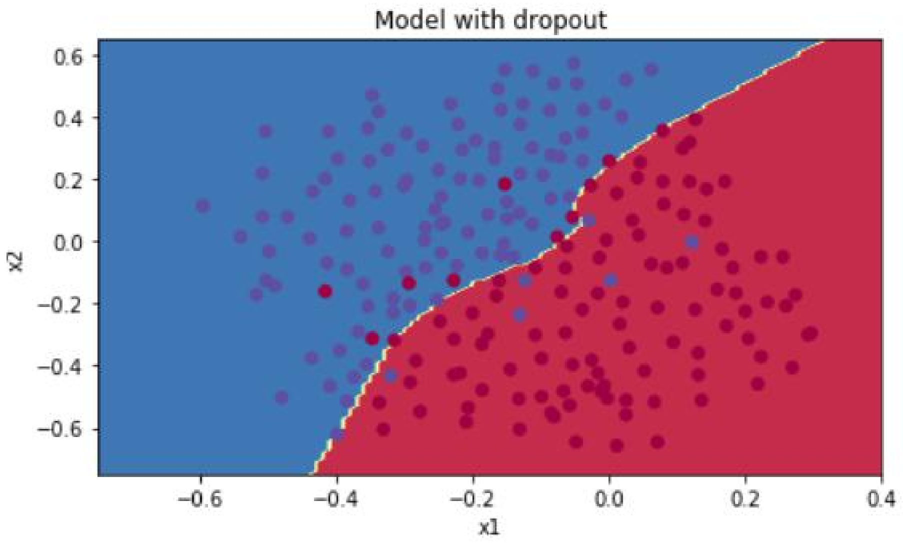
Expected Output:

dA1	[[0.36544439 0. -0.00188233 0. -0.17408748] [0.65515713 0. -0.00337459 0. -0.]]
dA2	[[0.58180856 0. -0.00299679 0. -0.27715731] [0. 0.53159854 -0. 0.53159854 -0.34089673] [0. 0. -0.00292733 0. -0.]]

实验结果如下：



```
On the train set:
Accuracy: 0.9289099526066351
On the test set:
Accuracy: 0.95
```



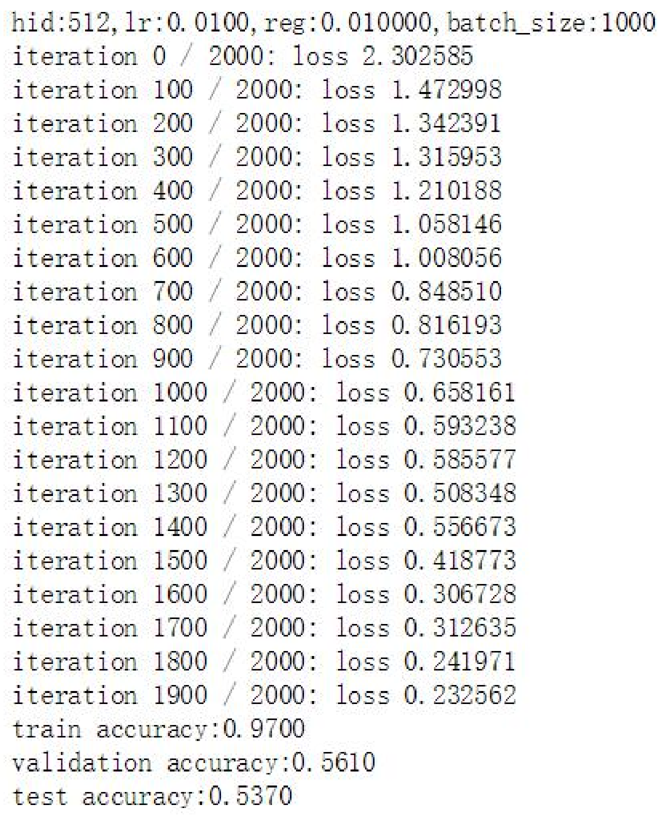
显然，使用了 dropout 技术以后，模型对测试集的准确率进一步提升了。

model	train accuracy	test accuracy
3-layer NN without regularization	95%	91.5%
3-layer NN with L2-regularization	94%	93%
3-layer NN with dropout	93%	95%

总的来说，以上正则化技术的运用有助于减少过拟合的情况，L2 regularization 和 dropout 是两个经常使用的有效的技术。

3. Improvement

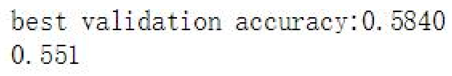
尝试用上述方法提升既有三层网络的效果，考虑到 L2 regularization 已经被运用，最终决定在既有模型中新增 **He initialization** 以及 **Adam** 来改进模型。（主要对 neural_net.py 进行操作）



```
hid:512,lr:0.0100,reg:0.010000,batch_size:1000
iteration 0 / 2000: loss 2.302585
iteration 100 / 2000: loss 1.472998
iteration 200 / 2000: loss 1.342391
iteration 300 / 2000: loss 1.315953
iteration 400 / 2000: loss 1.210188
iteration 500 / 2000: loss 1.058146
iteration 600 / 2000: loss 1.008056
iteration 700 / 2000: loss 0.848510
iteration 800 / 2000: loss 0.816193
iteration 900 / 2000: loss 0.730553
iteration 1000 / 2000: loss 0.658161
iteration 1100 / 2000: loss 0.593238
iteration 1200 / 2000: loss 0.585577
iteration 1300 / 2000: loss 0.508348
iteration 1400 / 2000: loss 0.556673
iteration 1500 / 2000: loss 0.418773
iteration 1600 / 2000: loss 0.306728
iteration 1700 / 2000: loss 0.312635
iteration 1800 / 2000: loss 0.241971
iteration 1900 / 2000: loss 0.232562
train accuracy:0.9700
validation accuracy:0.5610
test accuracy:0.5370
```

结果 loss 下降的很快，且训练准确率，验证准确率，测试准确率都得到了提升，以上方法的改进效果是显著的。

训练得出的最好的模型对测试集的准确率如下：
参数为 hidden_size=512 lr=0.005 reg=0.001 batch_size=500



```
best validation accuracy:0.5840
0.551
```

如果进一步调整的话，一定可以得到更好的结果。

结论分析与体会：
Stanford 的实验引导真的很棒，很多之前一知半解的神经网络相关知识在此次实验中都得到了解答。还有调参真是一个玄学的过程，不到最后你真的不知道你的 loss 会不会下降（有可能有一刻它突然就下降了），说实话我的耐心受到了莫大的挑战。Adam 的运用让我着实感受到了模型性能的显著提升，此类技术的熟悉一定会帮助我以后的实验的进行。

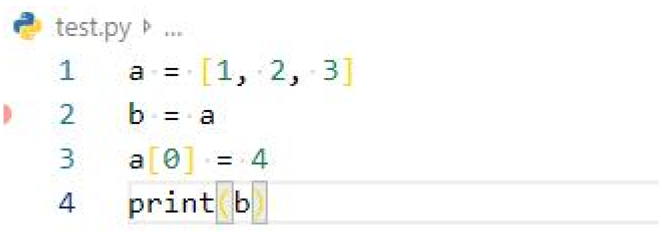
就实验过程中遇到和出现的问题，你是如何解决和处理的，自拟 1－3 道问答题：
1. 正确的引入 L2 regularization 的 loss 计算公式究竟是怎样的？
按照实验文件中展示的公式：

$$J = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)$$

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}}$$

正则项确实有一个λ/2m 系数的存在，代码中正则项也理应带上这个系数，然而事实上很多人的代码都缺少除以 m 的这一个操作，这也直接影响到反向传播相关的梯度计算。其实实际操作发现引不引入/m 都可以，因为λ/2m 本身你也可以把它看做另一个λ/2，只要调出让模型效果不错的合适的λ参数即可，有没有 m 无妨，这个地方并没有必要抓住不放。

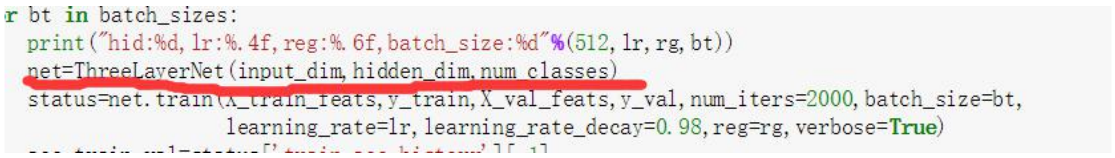
2. Python 中是不是会和 C++一样，假如有 b=a 这个命令，如果 a 在之后改变的话，再输出 b,那么 b 的结果是不是也会发生改变？
举个例子：



test.py ▸ ...

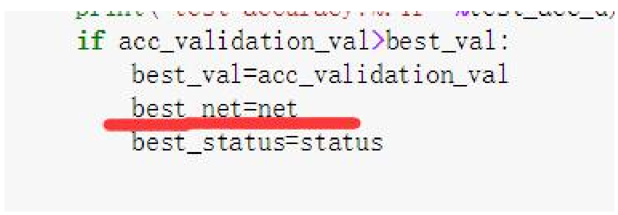
```
a = [1, 2, 3]
b = a
a[0] = 4
print(b)
```

很多人会认为输出的应该仍是[1,2,3]，然而事实上输出的是[4,2,3]，因为在这里 a 的首地址和 b 的首地址一样，改变了 a 的首地址指向的值 b 的首地址所指向的值理所当然会发生改变。这解释了实验中



```
r bt in batch_sizes:
  print("hid:%d, lr:%.4f, reg:%.6f, batch_size:%d"%(512, lr, rg, bt))
  net=ThreeLayerNet(input_dim, hidden_dim, num_classes)
  status=net.train(X_train_feats, y_train, X_val_feats, y_val, num_iters=2000, batch_size=bt,
                   learning_rate=lr, learning_rate_decay=0.98, reg=rg, verbose=True)
```

这行代码存在的原因。如果不在每次循环中新创一个 net 的话，



```
    if acc_validation_val>best_val:
        best_val=acc_validation_val
        best_net=net
        best_status=status
```

best_net 所代表的网络实际上是随着 net 的变化而变化的，无关前面的判断条件是否满足。这一点今后的实验一定要注意。