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：优化深层神经网络		学号：201918130222
日期：2021/10/7	班级： 智能	姓名： 魏江峰
Email：2257263015@qq.com		
实验目的： 掌握深层神经网络的一般优化方法		
实验软件和硬件环境： 硬件环境: 处理器：Intel core i7 9750-H 电脑：神州 z7m-ct7nk 软件环境: Pycharm 与 jupyter notebook		
实验原理和方法： 基于 Python 的科学计算库，实现神经网络并优化		

实验步骤：

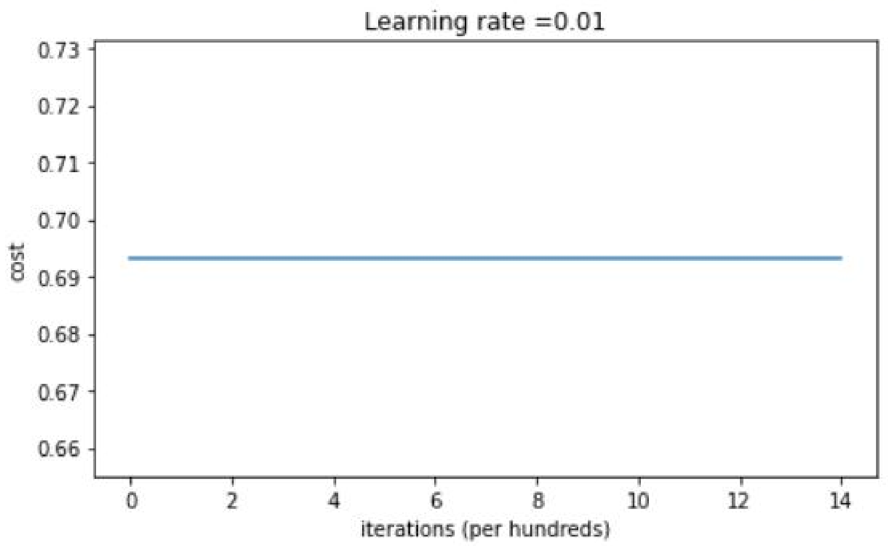
1，Initializaiton

#Zero initializaiton (weight is zero)

将权重矩阵设为零。

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

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

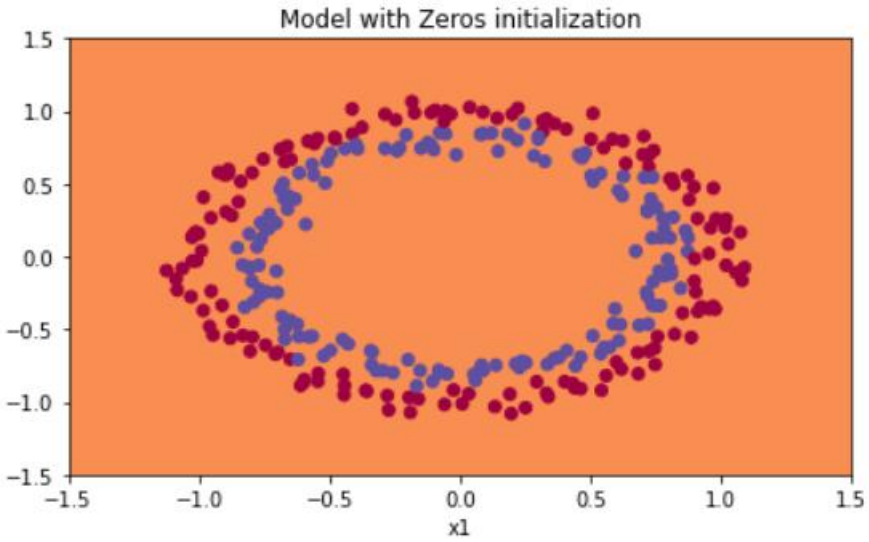


可以看到 loss 没有变化。显然 0*x+0=0，（0x）’=0，计算结果全是 0，梯度也全是 0，所以一致原地踏步。

```
predictions_train = [[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0]]
predictions_test = [[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

这种情况下，预测结果也全是零

决策边界完全包围数据集，全部都被预测为 0



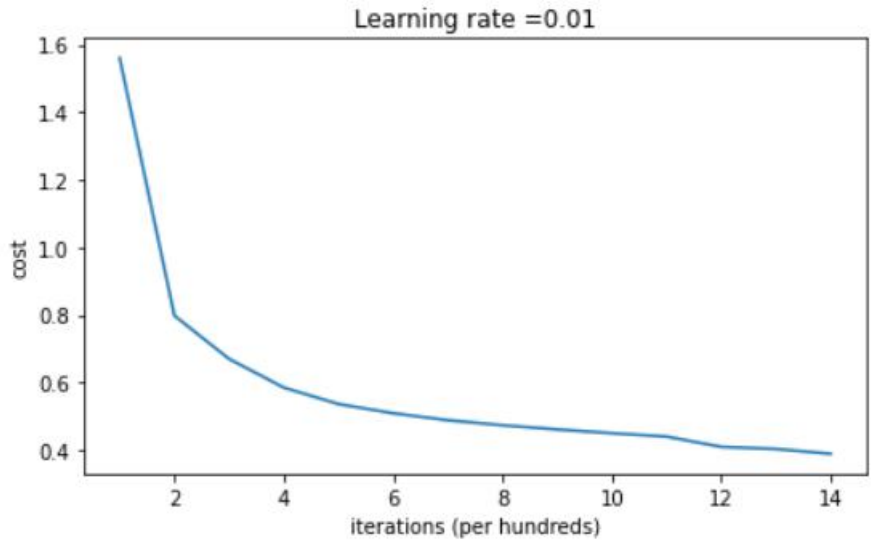
#Random initialization

使用随机数初始化 W 和 b：

parameters['W' + str(l)] =np.random.randn(layers_dims[l],

layers_dims[l-1])*8.0

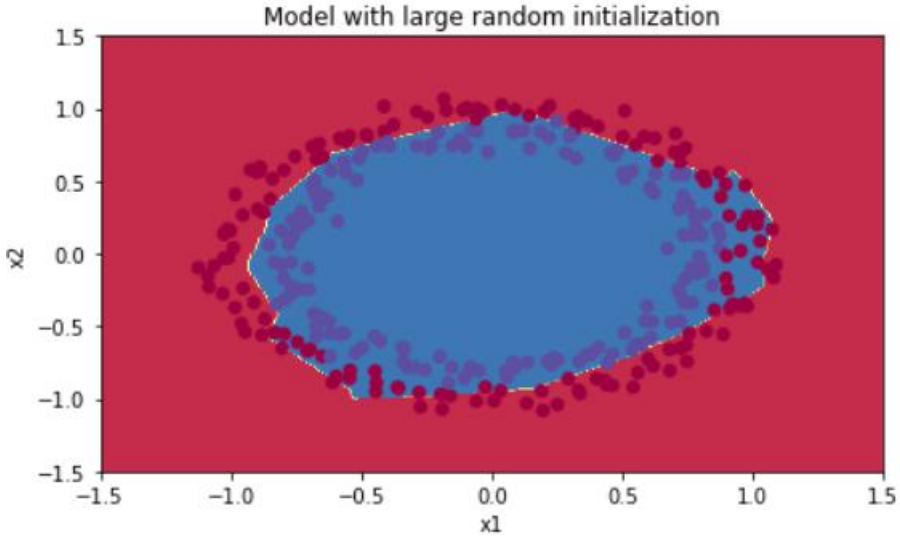
parameters['b' + str(l)] = np.random.randn(layers_dims[l],1)*8.0



可以看出，loss 逐渐下降。

```
On the train set:
Accuracy: 0.8733333333333333
On the test set:
Accuracy: 0.84
```

最后的准确率在 0.84 左右



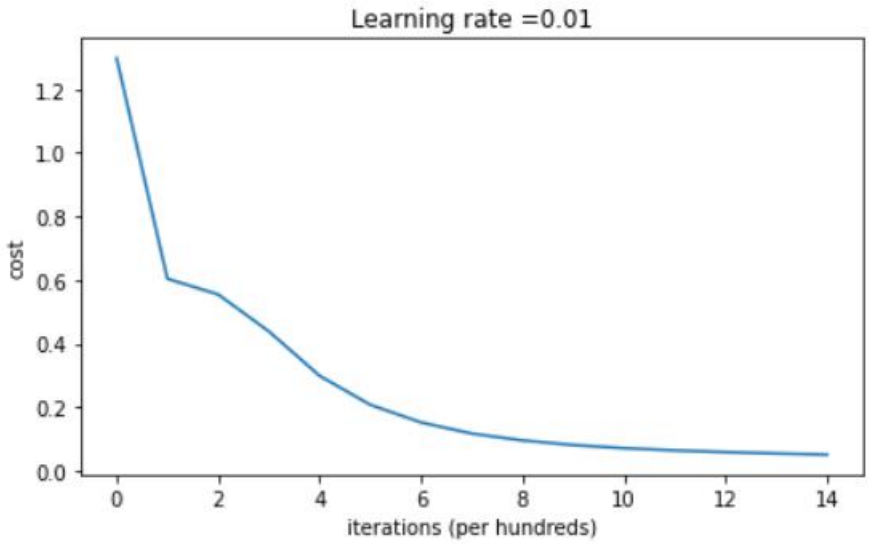
从决策边界也可以看出效果不错

#He initialization

由于我们使用了 ReLU 激活函数，PPT 中讲到了一个更好的优化方法：

$$W = \sqrt{\frac{2}{previous\ layer\ dimension}}$$

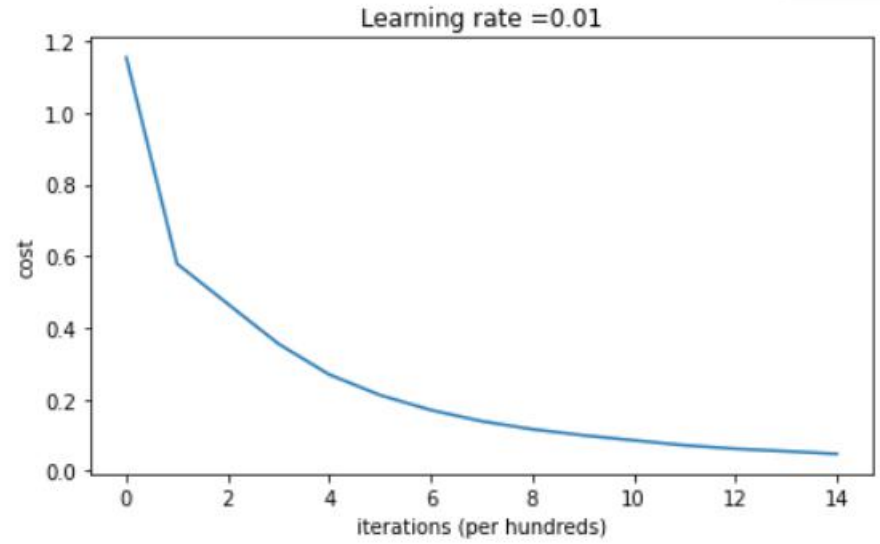
当对 b 也进行这样的初始化：




```
On the train set:
Accuracy: 0.99
On the test set:
Accuracy: 0.93
```

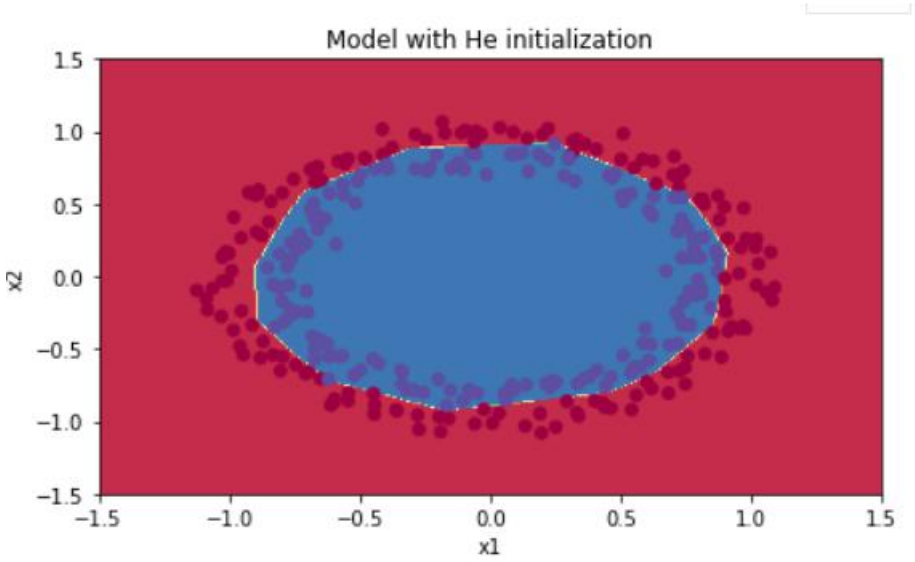
可以看出效果比 random 初始化好

b 全部设为 0：




```
On the train set:
Accuracy: 0.9966666666666667
On the test set:
Accuracy: 0.96
```

效果居然比 b 初始化更好，这似乎是一个问题



从决策边界也可以看出分类十分理想

2，Gradient checking

一维 gradient checking:这个好像没啥好说的……

N 维 gradient checking:

```
thetaplus = np.copy(parameters_values)                     # Step 1
thetaplus[i][0]    += epsilon                  # Step 2
J_plus[i], _ = forward_propagation_n(X,Y,vector_to_dictionary(thetaplus))
thetaminus = np.copy(parameters_values)                    # Step 1
thetaminus[i][0] -=epsilon                                 # Step 2
J_minus[i], _ = forward_propagation_n(X,Y,vector_to_dictionary(thetaminus))
 gradapprox[i] = (J_plus[i] - J_minus[i])/(2*epsilon)
  numerator = np.linalg.norm(grad-gradapprox)     # Step 1'
  denominator = np.linalg.norm(grad)+np.linalg.norm(gradapprox)
# Step 2'
  difference = numerator/denominator
```

```
Your backward propagation works perfectly fine! difference = 8.265882246781646e-09
```

可以看出结果基本正确。

3,Optimization

1，Gradient descent

计算参数关于全部训练集的梯度，并沿着梯度相反方向前进（一小步），循环往复，既可以到达极小值点

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

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

```
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]]
```

输出大致和期望一致

2，Mini-batch gradient descent

从训练集中选取一小部分，计算参数关于这一小部分的梯度并进行梯度下降。这种方法增加了迭代次数（小 patch 的梯度不一定是总的梯度），减小了每次迭代的计算量。

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]

mini_batch_X = shuffled_X[:,-(m-num_complete_minibatches*mini_batch_size)-1:-1]

mini_batch_Y = shuffled_Y[:,-(m-num_complete_minibatches*mini_batch_size)-1:-1]

3，Momentum

从上一部分我们直到 mini-batch gd 会产生振荡，为了减少这种振荡，就有了 Momentum 方法。

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

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

我们计算并更新了一个“速度”，使其不仅受当前梯度影响，也受之前的梯度影响。

可以使当前的梯度的决定性降低，从而降低振荡。

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)]

parameters["W" + str(l+1)] -= learning_rate*v["dW"+str(l+1)]

parameters["b" + str(l+1)] -= learning_rate*v["db"+str(l+1)]

4，Adam

Adam 适应性的参数估计。

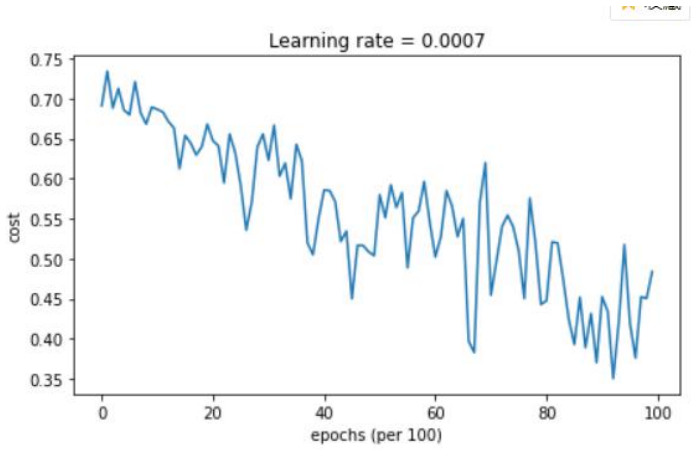
$$\begin{cases} v_{W^{[l]}} = \beta_1 v_{W^{[l]}} + (1-\beta_1)\frac{\partial J}{\partial W^{[l]}} \\ v^{corrected}_{W^{[l]}} = \frac{v_{W^{[l]}}}{1-(\beta_1)^t} \\ s_{W^{[l]}} = \beta_2 s_{W^{[l]}} + (1-\beta_2)(\frac{\partial J}{\partial W^{[l]}})^2 \\ s^{corrected}_{W^{[l]}} = \frac{s_{W^{[l]}}}{1-(\beta_2)^t}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

我们维护了 U 和 S 两个变量，和上面 Momentum 是差不多的意思。根据 U 和 S 计算出 corrected 的 U’和 S’,并用他们进行梯度的更新。

```
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)]



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)

 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)

  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)

   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)
```

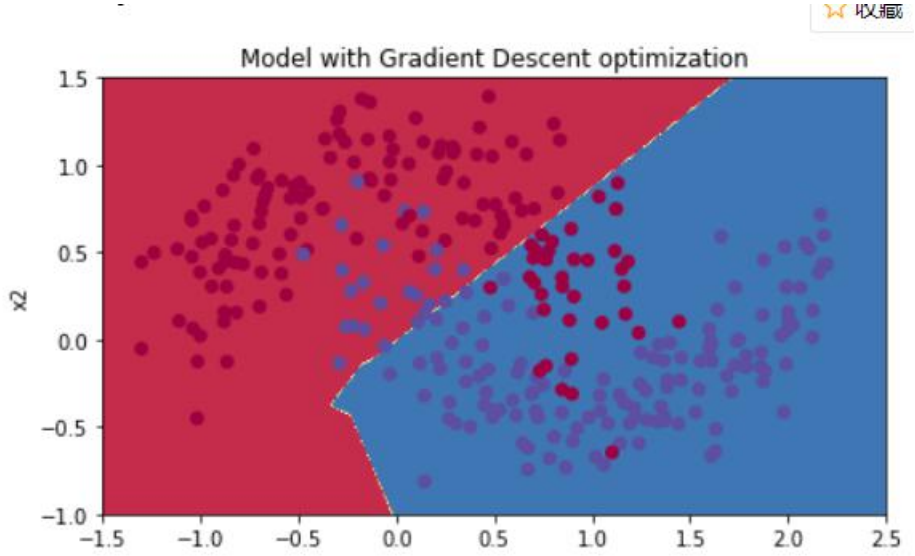
5 - Model with different optimization algorithms

5.1 Mini-batch gradient descent:



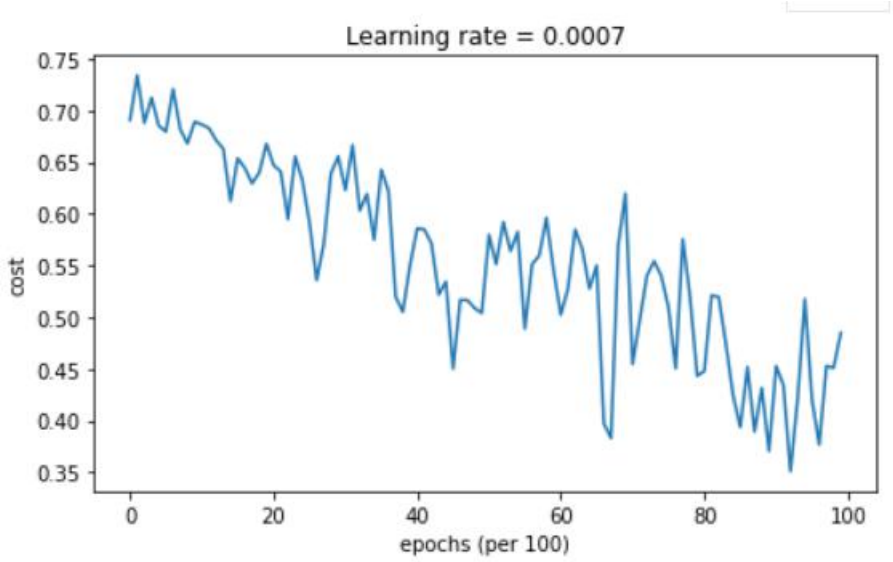
可以看出有明显的振荡

Accuracy: 0.7966666666666666

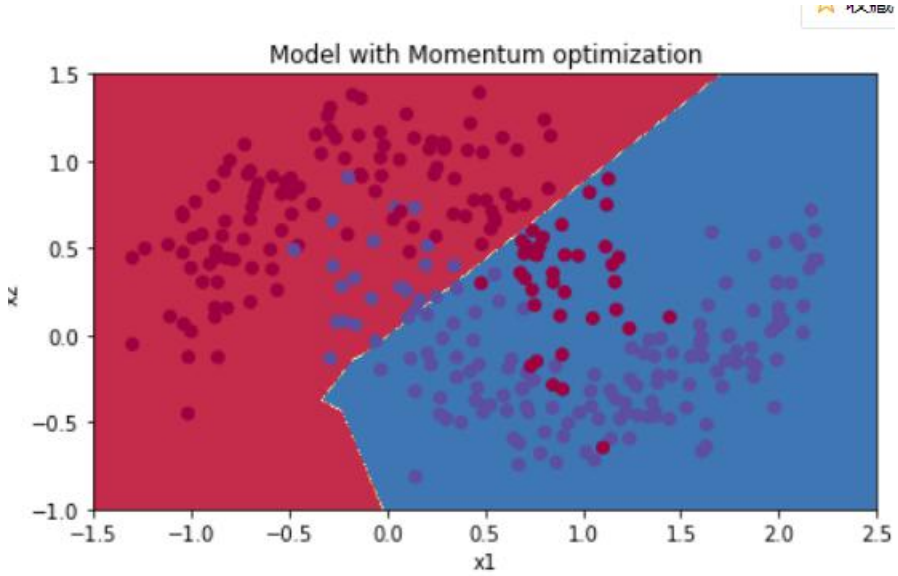


10000 次迭代下，准确率一般，决策边界不太对

5.2 Mini-batch gradient with momentum：

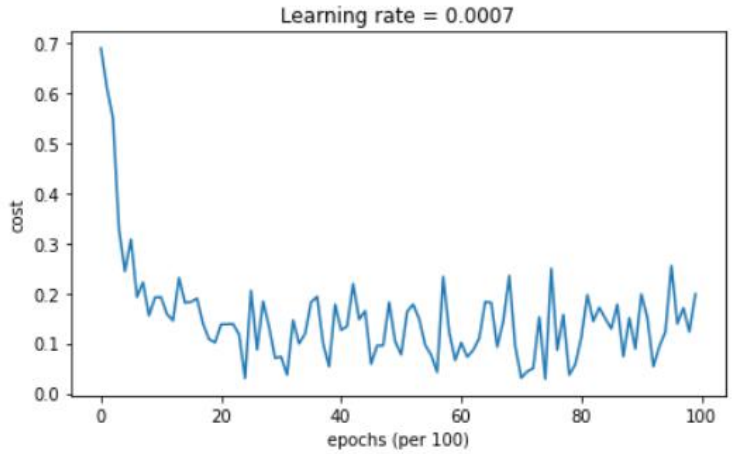


Accuracy: 0.7966666666666666

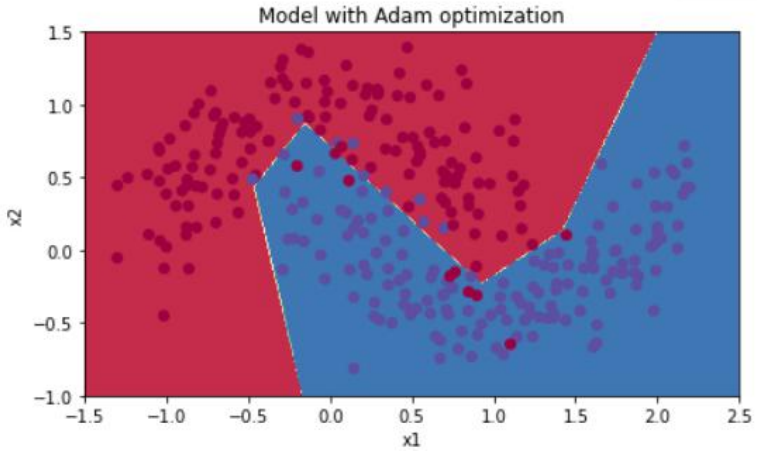


无论是结果还是优化过程都和 SGD 没有没有区别？？？？

5.3 Adam




```
Accuracy: 0.94
```

可以看出：

Adam 收敛速度大大加快，大概 2000 次迭代就达到了不错的效果。

准确率相较于上面的方法大大加快。

决策边界比较正确

结论分析与体会：

1，根据不同的激活函数选择不同的初始化策略可以大大提升模型性能。如 ReLU 选择 He initializaion 计算数值梯度逻辑相对简单，可以用来检验解析梯度的结果是否正确。

2，SGD 容易出现震荡的问题。Momentum 方法在数据集太简单了或者不合理的学习率情况下和 SGD 也没啥区别。

3，Adam 优化方法的性能最好。