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：Improving Deep Neural Networks		学号：201900130143
日期：10/7	班级： 智能	姓名： 吴家麒
Email：996362192@qq.com		

实验目的：

In this assignment you will master basic neural network adjustment skills and tryto improve deep neural networks: Hyperparameter tuning, Regularization and Optimization.

• this time you will be given three subtasks in folder "Improving Deep Neural Networks:Initialization, Gradient Checking, and Optimization".

实验软件和硬件环境：

Anaconda3 + JupyterNotebook

实验原理和方法：

1. 我们有时候实现完 backward propagation，我们不知道自己实现的 backward propagation 到底是不是完全正确，因此，通常要用梯度检验来检查自己实现的 bp 是否正确。梯度检验就是自己实现导数的定义，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。
2. 训练神经网络时，选择一个良好的初始化权重对于模型有着很大的帮助，能够加快梯度下降的收敛速度以及提高收敛的效果。
3. 参数更新的方法是决定模型训练效果的关键，我们尝试了几种常见的优化方法，来查看不同方法优化参数的特点与使用场景。

实验步骤：（不要求罗列完整源代码）

1. Gradient Checking

   一维梯度检验：

   假设一个 $J(\theta)=\theta x$，计算 forward_propagation，

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

```
J = 8
```

Expected Output:

** J **	8

再计算 backward_propagation，

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

```
dtheta = 2
```

Expected Output:

** dtheta **	2

进行梯度检验，计算公式如下：

- First compute "gradapprox" using the formula above (1) and a small value of ε. Here are the Steps to follow:
  1. $\theta^{+} = \theta + \varepsilon$
  2. $\theta^{-} = \theta - \varepsilon$
  3. $J^{+} = J(\theta^{+})$
  4. $J^{-} = J(\theta^{-})$
  5. $gradapprox = \frac{J^{+} - J^{-}}{2\varepsilon}$
- Then compute the gradient using backward propagation, and store the result in a variable "grad"
- Finally, compute the relative difference between "gradapprox" and the "grad" using the following formula:

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

得到结果：

```
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

N 维梯度检验：

```
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 = 0.2850931566540251
```

Expected output:

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

发现 backward propagation 的计算存在问题，检查后发现反向传播函数中 db1 的值错误。

修改后结果为：

```
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 = 1.1885552035482147e-07
```

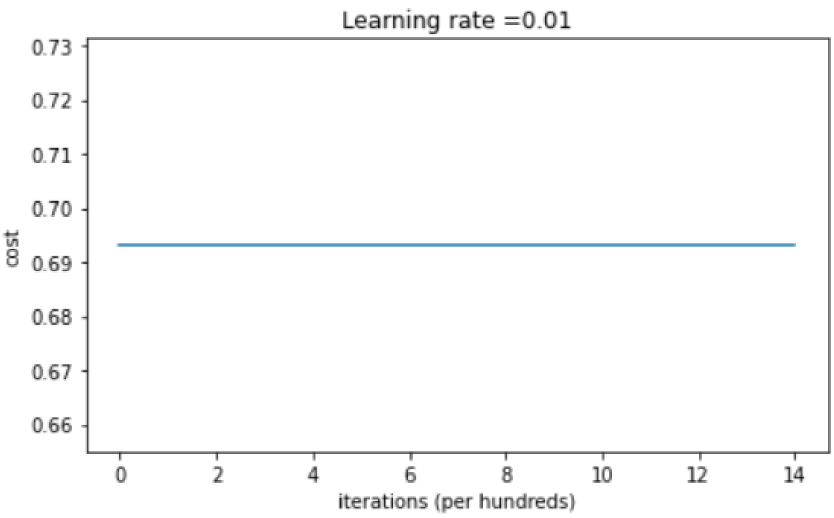
2. Initialization

初始化参数为 0：

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

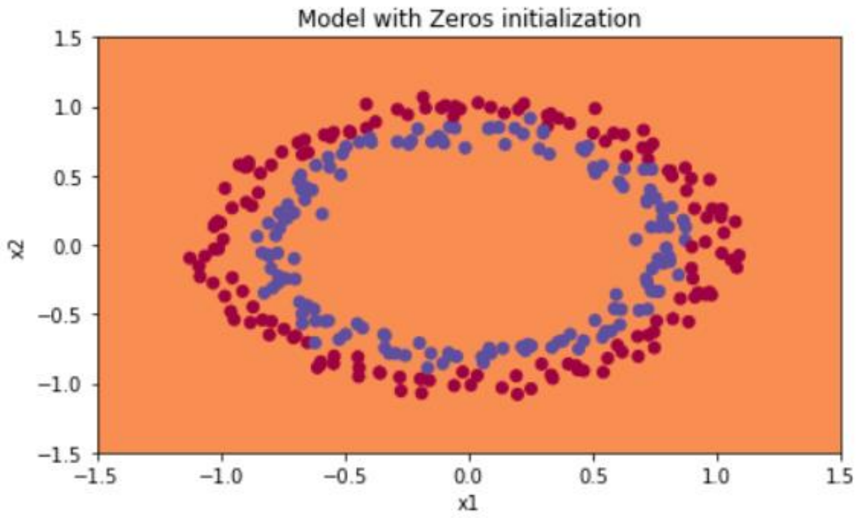
```
W1 = [[0. 0. 0.]
 [0. 0. 0.]]
b1 = [[0.]
 [0.]]
W2 = [[0. 0.]]
b2 = [[0.]]
```

计算模型训练结果：




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

对模型进行可视化：

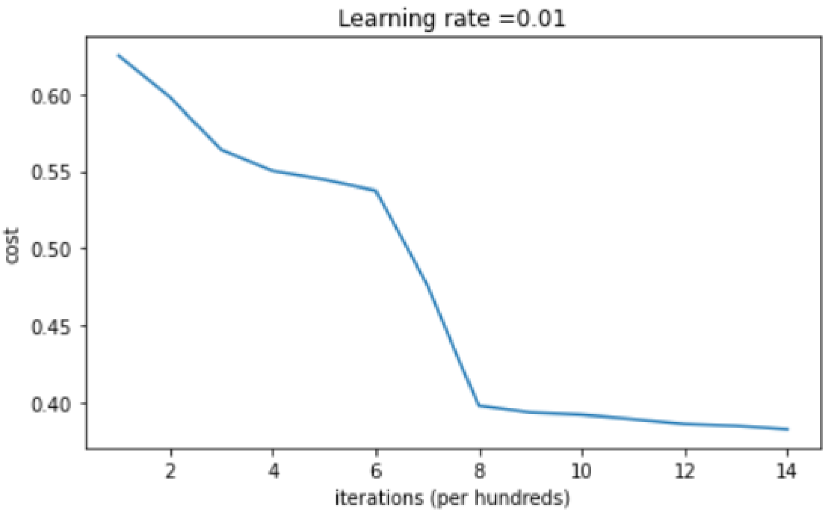


随机初始化参数：

```
]: 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"]))
```

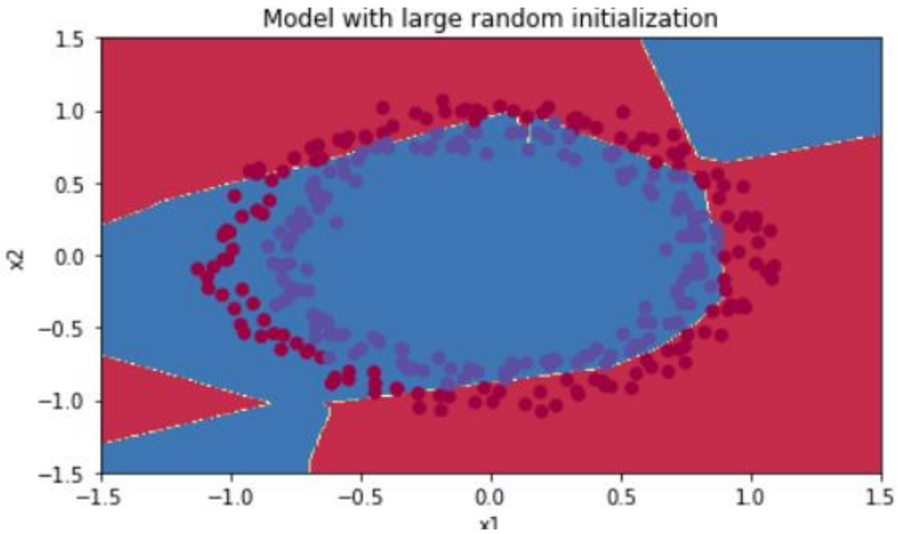
```
W1 = [[ 17.88628473   4.36509851   0.96497468]
 [-18.63492703  -2.77388203  -3.54758979]]
b1 = [[0.]
 [0.]]
W2 = [[-0.82741481 -6.27000677]]
b2 = [[0.]]
```

得到模型的训练结果：




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

模型可视化：



可以发现随机参数的分离效果并不好。

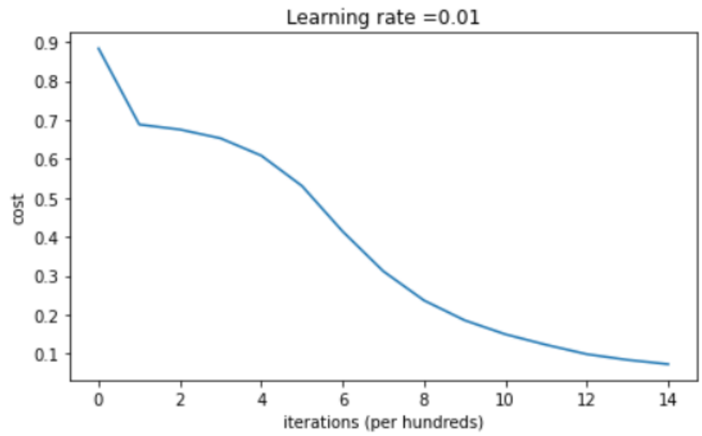
He 法初始化参数：

```
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"]))
```

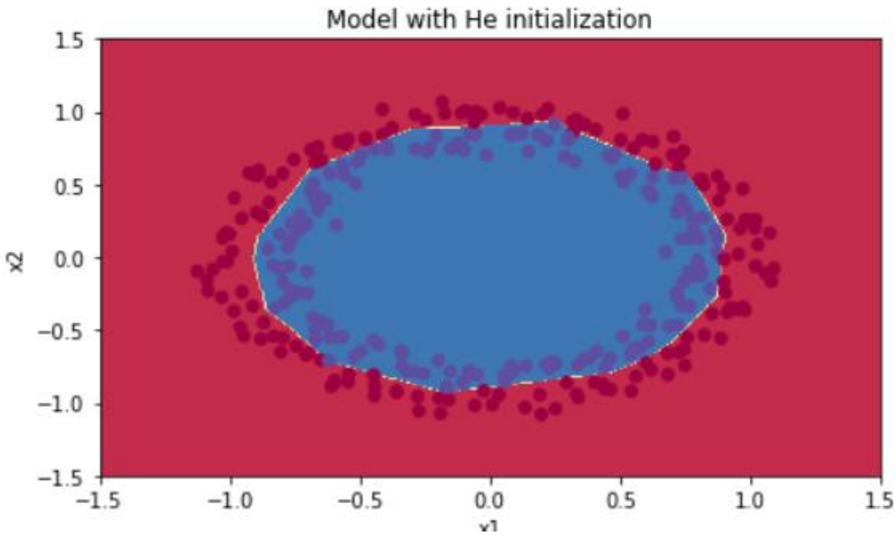
```
W1 = [[ 1.78862847  0.43650985]
 [ 0.09649747 -1.8634927 ]
 [-0.2773882  -0.35475898]
 [-0.08274148 -0.62700068]]
b1 = [[0.]
 [0.]
 [0.]
 [0.]]
W2 = [[-0.03098412 -0.33744411 -0.92904268  0.62552248]]
b2 = [[0.]]
```

得到模型训练结果：

```
Cost after iteration 0: 0.8830537463419761
Cost after iteration 1000: 0.6879825919728063
Cost after iteration 2000: 0.6751286264523371
Cost after iteration 3000: 0.6526117768893807
Cost after iteration 4000: 0.6082958970572937
Cost after iteration 5000: 0.5304944491717495
Cost after iteration 6000: 0.4138645817071793
Cost after iteration 7000: 0.3117803464844441
Cost after iteration 8000: 0.23696215330322556
Cost after iteration 9000: 0.18597287209206828
Cost after iteration 10000: 0.15015556280371808
Cost after iteration 11000: 0.12325079292273548
Cost after iteration 12000: 0.09917746546525937
Cost after iteration 13000: 0.08457055954024274
Cost after iteration 14000: 0.07357895962677366
```



模型可视化：



可以发现 He 初始化的模型可以在少量迭代中很好地分离蓝点和红点。

3. Optimization Methods

我们将对几种常见的深度学习参数优化方法进行尝试。

梯度下降法：

```
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]]
```

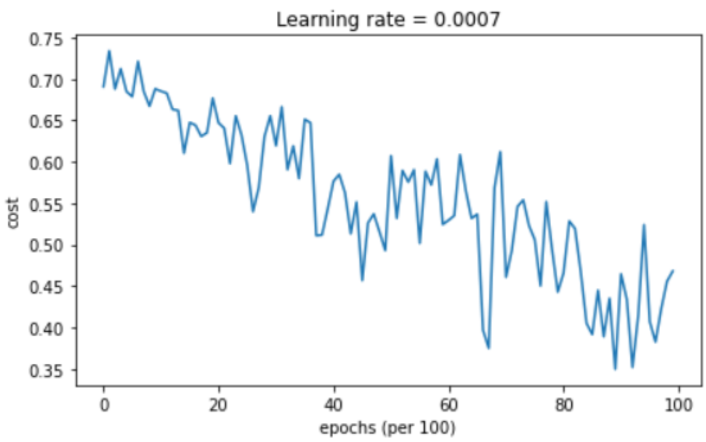
Mini-Batch Gradient descent:

计算结果:

```
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 ]
```

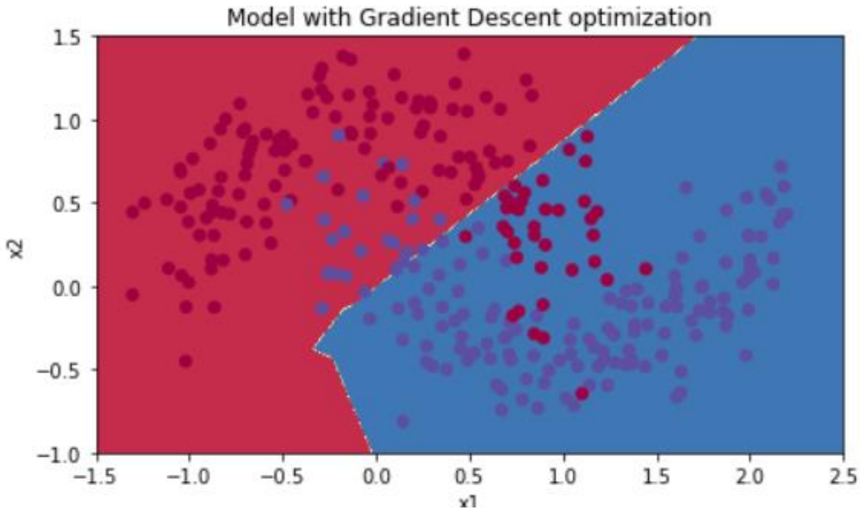
模型训练效果:

```
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: 0.7966666666666666
```

可视化:



Momentum:
由于 mini-batch gradient descent 在只看了一个例子的子集后进行参数的更新，更新的方向有一定的方差，所以 mini batch 所采取的路径会“振荡”向收敛。利用 Momentum 可以减少这些振荡。
我们可以将梯度的方向保存在变量 v 中，将其视为梯度下降的，根据梯度的方向来增加速度（和 momentum）。

初始化：

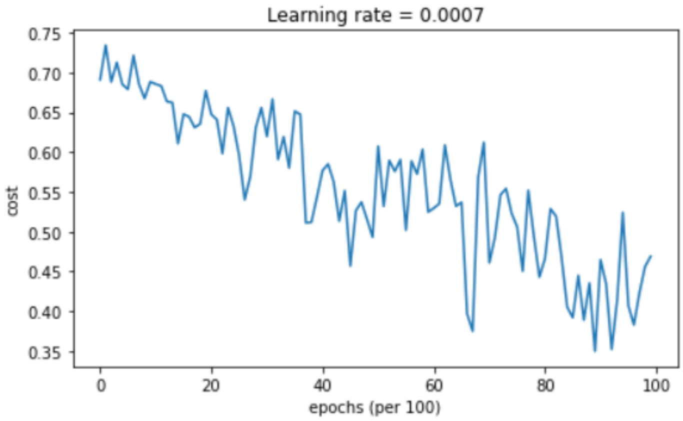
```
v["dW1"] = [[0.  0.  0.]
 [0.  0.  0.]]
v["db1"] = [[0.]
 [0.]]
v["dW2"] = [[0.  0.  0.]
 [0.  0.  0.]
 [0.  0.  0.]]
v["db2"] = [[0.]
 [0.]
 [0.]]
```

参数更新：

```
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]]
```

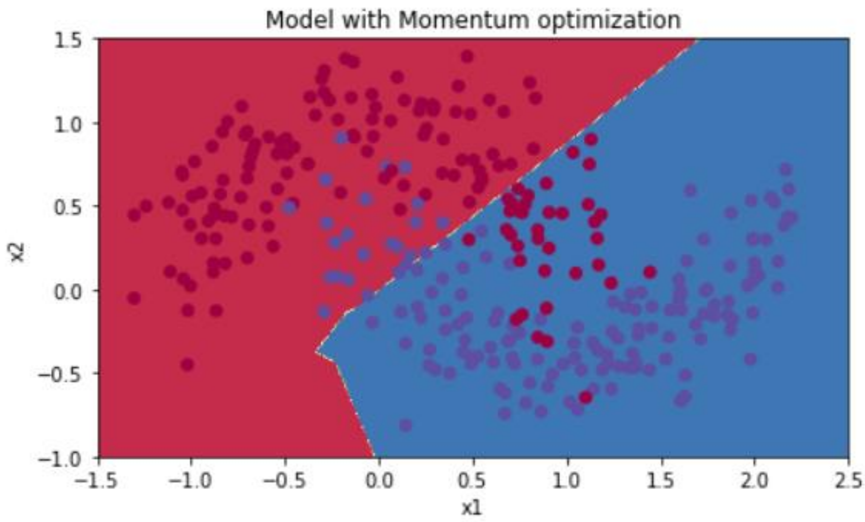
模型训练效果：

```
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: 0.7966666666666666
```

可视化：



Adam 法：

初始化参数，

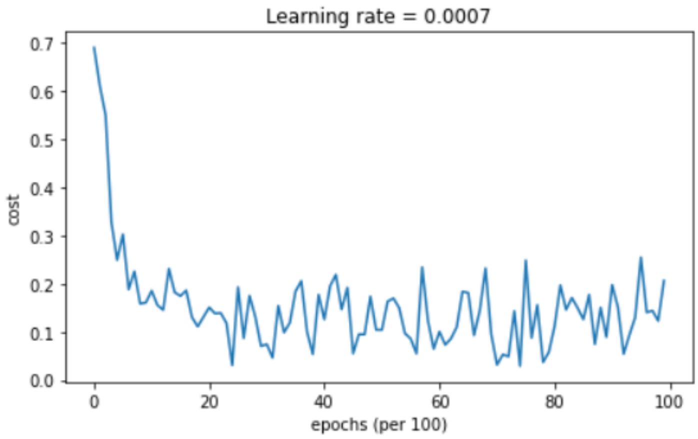
```
v["dW1"] = [[0.  0.  0.]
 [0.  0.  0.]]
v["db1"] = [[0.]
 [0.]]
v["dW2"] = [[0.  0.  0.]
 [0.  0.  0.]
 [0.  0.  0.]]
v["db2"] = [[0.]
 [0.]
 [0.]]
s["dW1"] = [[0.  0.  0.]
 [0.  0.  0.]]
s["db1"] = [[0.]
 [0.]]
s["dW2"] = [[0.  0.  0.]
 [0.  0.  0.]
 [0.  0.  0.]]
s["db2"] = [[0.]
 [0.]
 [0.]]
```

参数更新过程：

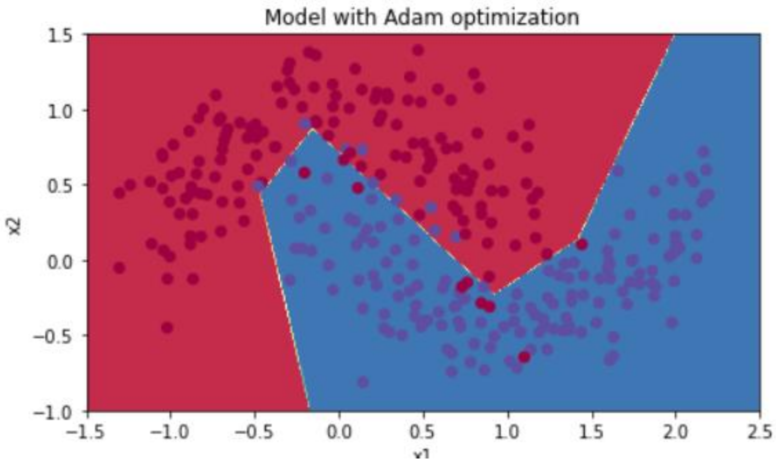
```
W1 = [[ 1.63179046 -0.61920151 -0.53561684]
 [-1.08041371  0.85796254 -2.29409361]]
b1 = [[ 1.75225686]
 [-0.75376181]]
W2 = [[ 0.32648419 -0.25681547  1.46955303]
 [-2.05269562 -0.31497211 -0.37660926]
 [ 1.14121453 -1.09244618 -0.16498312]]
b2 = [[-0.88530351]
 [ 0.03476866]
 [ 0.57537012]]
v["dW1"] = [[-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]]
s["dW1"] = [[0.00121136 0.00131039 0.00081287]
 [0.0002525  0.00081154 0.00046748]]
s["db1"] = [[1.51020075e-05]
 [8.75664434e-04]]
s["dW2"] = [[7.17640232e-05 2.81276921e-04 4.78394595e-04]
 [1.57413361e-04 4.72206320e-04 7.14372576e-04]
 [4.50571368e-04 1.60392066e-07 1.24838242e-03]]
s["db2"] = [[5.49507194e-05]
 [2.75494327e-03]
 [5.50629536e-04]]
```

模型训练结果：

```
Cost after epoch 0: 0.690552
Cost after epoch 1000: 0.185514
Cost after epoch 2000: 0.150822
Cost after epoch 3000: 0.074445
Cost after epoch 4000: 0.125931
Cost after epoch 5000: 0.104227
Cost after epoch 6000: 0.100425
Cost after epoch 7000: 0.031602
Cost after epoch 8000: 0.111753
Cost after epoch 9000: 0.197666
```



可视化：



结论分析与体会：

1. 通过本次实验，熟悉了通过梯度检验来确定反向传播算法是否得到了良好的执行。
2. 对于三种常见的模型参数初始化效果有了明确的认识，学习到了好的初始化参数对于模型训练的重要帮助。

   三种方式定义参数的效果：

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

3. 对于几种常见的深度学习优化方式进行了学习，对它们的实现方法有了较好的理解。

几种优化方法的比较：

optimization method	**accuracy**	**cost shape**
Gradient descent	79.7%	oscillations
Momentum	79.7%	oscillations
Adam	94%	smoother

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. 在训练时出现了不太正确的模型训练结果：

   需要注意的是，模型中的 weight 和 bias 都作为参数，更新的方式是一样，要注意两个参数更新时的一致性。