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：Initialization, Gradient Checking and Optimization		学号：202000130047
日期：2022/9/28	班级： 智能 20	姓名： 夏再禹
Email：842649082@qq.com		

实验目的：

将三个小实验的代码补充完整，通过实验理解初始化、反向传播与梯度近似计算、梯度检验，以及对各种优化算法如 SGD、Momentum、Adam 等进行实验与比较，了解他们的不同效果与差异。

实验软件和硬件环境：

硬件：联想小新笔记本，cpu: i5

软件：anaconda, python3.7

实验原理和方法：

1. Initialization

（1） 全为 0 的初始化没有打破参数的对称性，优化到最后同类型的参数的值都相同，效果很差。

（2） 单纯的用同一分布随机初始化参数，会导致 x 在前向传播过程中的每一层的分布可能会有较大变化，最后的效果可能不好。

（3） 对使用 ReLu 的神经网络，最好采用"He Initialization"进行初始化，即每一层的参数在随机初始化后乘以 $\sqrt{\frac{2}{\text{dimension of the previous layer}}}$。这可以让初始情况下 x 在前向传播过程中的每一层的分布不会变得极端。

2. Gradient Checking

（1）反向传播

反向传播的原理为链式求导法则，前向传播是从 x 经过层层神经网络计算出 y 的过程；而反向传播与前向传播的计算顺序相反，从 loss 开始，逐个往回计算梯度，这是因为计算第 i 层的梯度时需要第 i+1 层的梯度值。

（2）梯度近似计算

从梯度的定义出发，我们有：

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

于是我们只需要取一个很小的ε的值，然后计算参数加上ε后与减去ε后的值的情况下的损失函数的值之差，再除以两倍的ε，即可估算出梯度的值。

（3）梯度检验

我们在神经网络中一般采用反向传播计算梯度，如果想要对反向传播过程进行检验，可以用梯度近似计算方法算出一个梯度，再与反向传播算出的梯度进行比较，看他们是不是十分相近。

3. Optimization

（1）GD、SGD 与 Mini-Batch 梯度下降

GD 即每次使用整个数据集计算梯度，然后再更新梯度。而 SGD 每次采用一个数据来计算梯度，并更新梯度。

在数据集很大的时候，我们可能没有足够的时间与计算资源来进行 GD，而 SGD 的优化速度将更快，但因每次只采用一部分数据进行梯度下降，其优化路径将有些 zigzag。

Mini-Batch 梯度下降每次采用一部分的数据来计算梯度，其与 SGD 的区别是其通过小批量梯度下降，遍历小批量，而不是遍历各个训练示例。使用 mini-batch 批处理通常可以加快优化速度

（2） Momentum

因为 Mini-Batch 梯度下降仅在看到示例的子集后才进行参数更新，所以更新的方向具有一定的差异，因此小批量梯度下降所采取的路径将“朝着收敛”振荡。利用冲量则可以减少这些振荡。冲量考虑了过去的梯度以平滑更新。通过将先前梯度的“方向”存储在变量中，这也就是先前步骤中梯度的指数加权平均值，与本次的梯度的线性加权作为每次参数的改变量。

通过使用冲量，优化路径的振荡将减少，速度将加快。

（3）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 的优点 Adam 算法同时获得了 AdaGrad 和 RMSProp 算法的优点,其不仅如 RMSProp 算法那样基于一阶矩均值计算适应性参数学习率，它同时还充分利用了梯度的二阶矩均值（即有偏方差/uncentered variance）。具体来说，算法计算了梯度的指数移动均值（exponential moving average），超参数 beta1 和 beta2 控制了这些移动均值的衰减率。

Adam 算法将比 Momentum 算法有更少的振荡，且更容易跳出局部最优。

实验步骤：（不要求罗列完整源代码）

1. Initialization

(1) Zero initialization

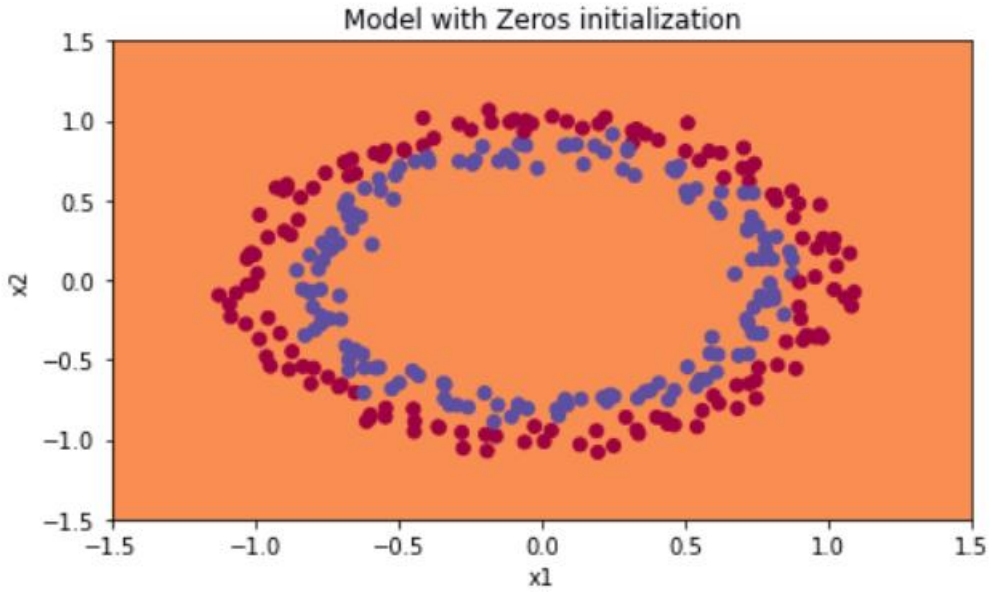
我们将所有参数都赋值为 0 即可。

```
    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
```

初始化的参数为：

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

训练结果为：



可见，由于没有打破参数的对称性，优化到最后同类型的参数的值都相同，效果很差，最终将所有点都预测到了同一个类别。

（2）Random initialization

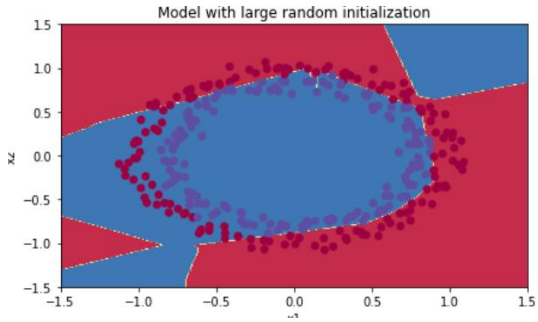
每一层的参数都用相同的分布进行初始化：

```
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 ###
```

初始化的结果为：

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

训练结果为：



可见，其打破了参数的对称性，但效果一般，还是不够好。

（3）He initialization

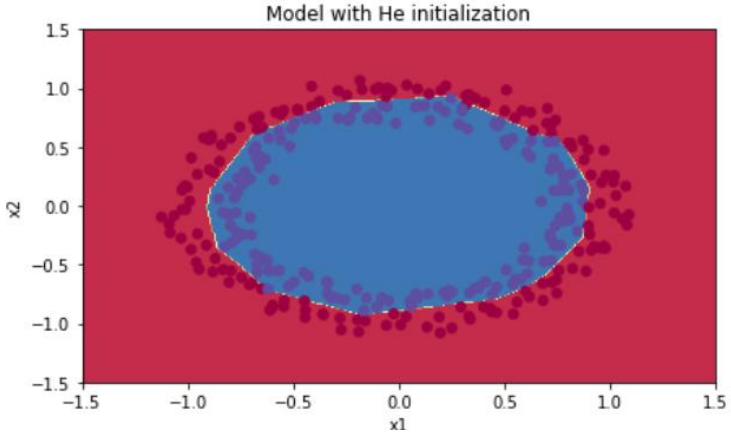
每一层的参数在随机初始化后乘以$\sqrt{\frac{2}{\text{dimension of the previous layer}}}$：

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

其初始化的参数为：

```
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.]]
```

训练结果为：



可见，He initialization 获得了最好的效果。

2. Gradient Checking

(1)1 维梯度检验

一维的情况较为简单，就是一个简单的线性变化，补充代码如下：

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

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

对于梯度检验的过程，按照梯度近似算法，一步步算出 J_plus、J_minus，再算出估计的梯度，然后与我们计算出的梯度进行比较。

```
### START CODE HERE ### (approx. 5 lines)
thetaplus = theta+epsilon                                    # Step 1
thetaminus = theta-epsilon                                   # Step 2
J_plus = forward_propagation(x,thetaplus)
J_minus = forward_propagation(x,thetaminus)
gradapprox = (J_plus-J_minus)/(2*epsilon)
### END CODE HERE ###

# Check if gradapprox is close enough to the output of backward_
### 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)
denominator = np.linalg.norm(grad)+np.linalg.norm(gradapprox)
difference = numerator/denominator
### END CODE HERE ###
```

梯度检验的结果为：

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

说明梯度计算正确。

（2）n 维梯度检验

梯度检验过程其实与一维的类似，这里要注意使用深拷贝：

```
### START CODE HERE ### (approx. 3 lines)
thetaplus = np.copy(parameters_values)                                #
thetaplus[i][0] = thetaplus[i][0]+epsilon                            # S
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_minu
### START CODE HERE ### (approx. 3 lines)
thetaminus = np.copy(parameters_values)
thetaminus[i][0] = thetaminus[i][0]-epsilon                          # 
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 ###
```

最后检验的结果为：

```
There is a mistake in the backward propagation! difference = 0.2850931567761623
```

说明反向传播过程有误。

（3）修正反向传播过程

我们重新定义反向传播函数，根据推导，我发现 dw2 与 db1 的计算公式有误，修改他们如下：

```
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)
```

然后再进行梯度检验，可得到：

```
difference = gradient_check_n(parameters, gradients, X, Y,epsilon = 1e-6)
```

```
Your backward propagation works perfectly fine! difference = 8.265882247803851e-09
```

说明此时的梯度计算正确。

3. Optimization Methods

补充梯度下降的代码，使用 GD 的基本更新公式即可：

```
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 ###
```

补充 mini-batch 的代码，首先将数据集随机划分为一个一个 batch

```
### 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 ###
```

```
### START CODE HERE ### (approx. 2 lines)
mini_batch_X = shuffled_X[:, num_complete_minibatches * mini_batch_size : m]
mini_batch_Y = shuffled_Y[:, num_complete_minibatches * mini_batch_size : m]
### END CODE HERE ###
```

划分结果为：

```
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 ]
```

补充 Momentum 的代码：

首先对冲量进行初始化：

```
### START CODE HERE ### (approx. 2 lines)
v["dW" + str(l+1)] = np.zeros(parameters['W' + str(l+1)].shape)
v["db" + str(l+1)] = np.zeros(parameters['b' + str(l+1)].shape)
### END CODE HERE ###
```

初始化输出为：

```
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.]]
```

然后采用 Momentum 的更新法则，补充参数更新的代码：

```
### 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 ###
```

结果为

```
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]]
```

补充 Adam 的代码：

对一阶二阶梯度矩均值都进行初始化：

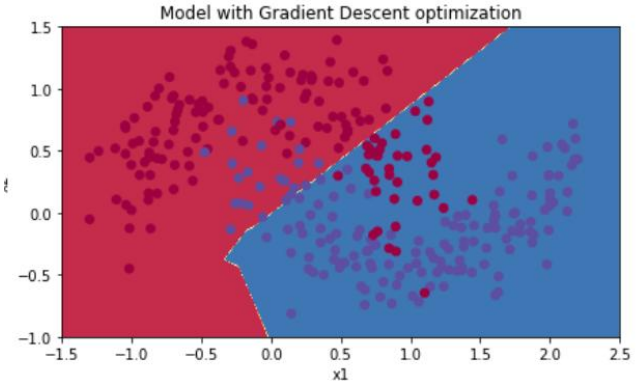
```
### START CODE HERE ### (approx. 4 lines)
    v["dW" + str(l+1)] = np.zeros(parameters["W" + str(l+1)].shape)
    v["db" + str(l+1)] = np.zeros(parameters["b" + str(l+1)].shape)
    s["dW" + str(l+1)] = np.zeros(parameters["W" + str(l+1)].shape)
    s["db" + str(l+1)] = np.zeros(parameters["b" + str(l+1)].shape)
### END CODE HERE ###
```

然后采用 Adam 的更新法则，补充参数更新的代码：

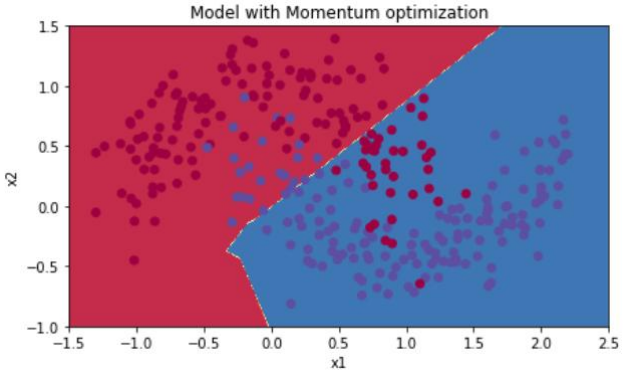
```
### START CODE HERE ### (approx. 2 lines)
v["dW" + str(l+1)] = beta1*v["dW" + str(l + 1)] +(1-beta1)*grads['dW' + str(l+1)]
v["db" + str(l+1)] = beta1*v["db" + str(l + 1)] +(1-beta1)*grads['db' + str(l+1)]
### END CODE HERE ###

# Compute bias-corrected first moment estimate. Inputs: "v, beta1, t". Output: "v_correct
### 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_co
### 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 ###
```

比较不同优化算法的结果：

GD:



Momentum:



Adam:

可见上述三种优化算法中，adam 优化算法的效果最好。

结论分析与体会：

首先对参数初始化进行了实验，通过实验验证，全初始化为 0 未能打破参数对称性，而对使用 ReLu 的神经网络，采用"He Initialization"进行初始化的效果最好。

然后对于梯度检验，实验了计算数值梯度的过程，也就是近似梯度，同时通过实验检验出了一个反向传播的错误，并进行了修改。

对于优化算法，实验完成了四种优化算法，通过代码懂得了对这些优化算法的具体实现，增强了对他们的原理的理解，同时通过实验证明，adam 优化算法的效果确实好很多，因此一般来说我们都可以采用 adam 优化器。

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. Gradient Checking 中的 n 维梯度检验与答案对不上

解决：通过输出中间结果，发现我改变了 parameters_values 的值，而这应该是不能变的。进而发现这是因为我直接用等号赋值，这其实是一个引用，而不是拷贝，因此我将 thetaplus = parameters_values 改为 thetaplus = np.copy(parameters_values)后问题即解决。