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：神经网络的基本训练		学号：201900150221
日期：10.7	班级： 19 智能	姓名： 张进华
Email：zjh15117117428@163.com		

实验目的：

掌握基本的神经网络调整技巧并尝试，改进深度神经网络：超参数调整、正则化和优化，共三个子任务：初始化、梯度检查和优化

实验软件和硬件环境：

Visual studio Code python 3.9.7

Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz

实验原理和方法

1.完成三个初始化，梯度计算，和最优化三个文件，分别实现了神经网络中最常用的集中优化神经网络的方式。

2.数据是直接用的是 sklearn 的专有的数据。每个都有对应的 py 文件，这个文件里面的东西不需要进行修改， 直接使用即可。

3.初始化，用于初始化权重，即神经元最开始的 w 的值，方便计算出不同的东西。

4.梯度检查，用于验证反向传播是否有效，正确，也就是在训练模型的时候，是否对 dw 和 db 的值更新正确。

5.优化，是用于加快函数计算速度，比如调整学习率之类的操作来进行。

实验步骤

步骤一 Initialization

实现权重初始化

1.Neural Network model

Zeros 初始化-- 在输入参数中设置 initialization = "zeros"

随机初始化-- 在输入参数中设置 initialization = "random" 将权重初始化为大的随机值

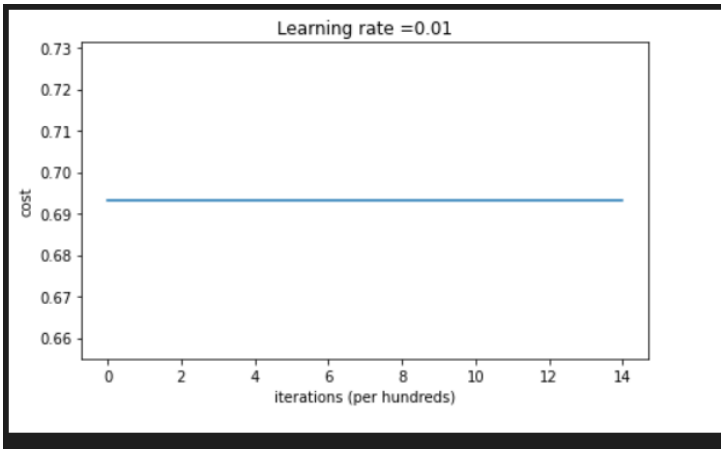
He 初始化 -- 在输入参数中设置 initialization = "he" 将权重初始化为根据 He 等人在 2015 年发表的论文缩放的随机值

2 .Zero initialization

实现函数以将所有参数初始化为零,不能很好地工作，因无法“破坏对称性”

```
    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
```

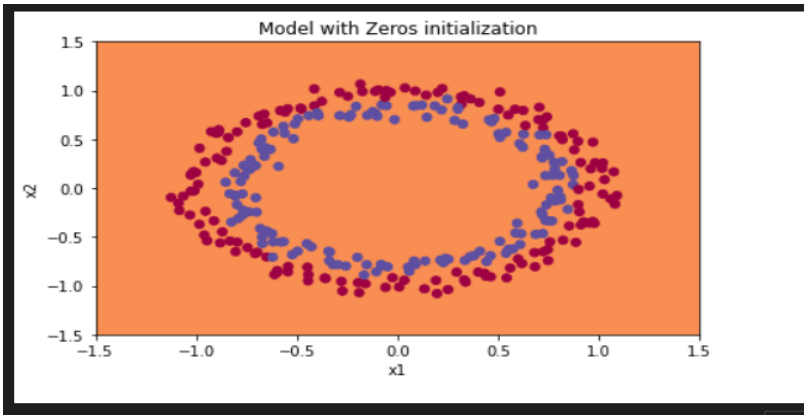
使用这些参数来训练模型



图中我们可以看到学习率一直没有变化，结论为模型根本没有学习，查看预测结果：

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

可见算法性功能比较差，查看预测和决策边界



分类失败，该模型预测每个都为 0。零初始化都会导致神经网络无法打破对称性，最终导致的结果就是无论网络有多少层，最终只能得到和 Logistic 函数相同的效果

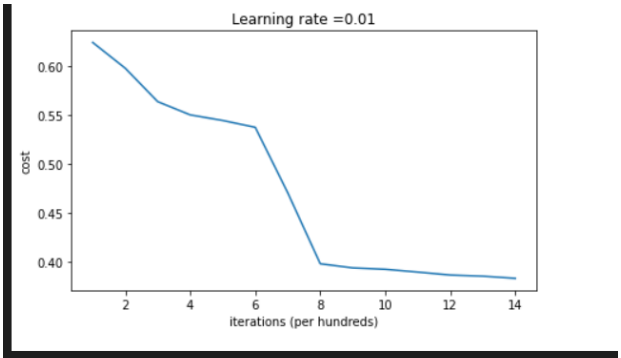
3. Random initialization

为了打破对称性，可以随机地把参数赋值。在随机初始化之后，每个神经元可以开始学习其输入的不同功能

```
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 #使用10倍缩放
    parameters['b' + str(l)] = np.zeros((layers_dims[l], 1))
    ### END CODE HERE ###

return parameters
```

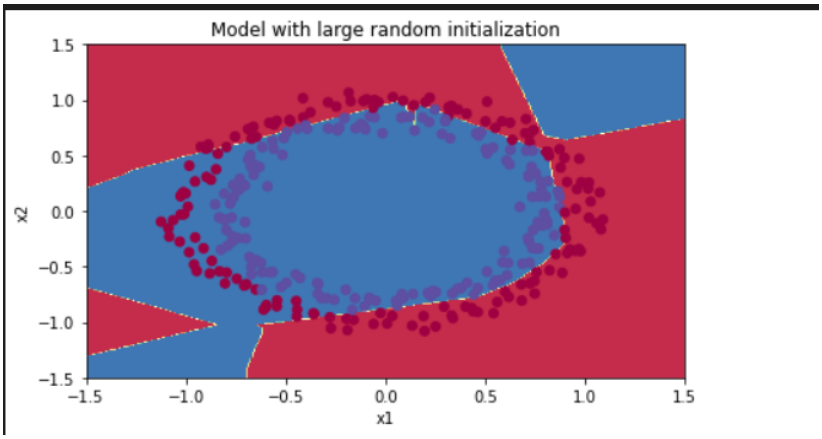
0.3s

在得到参数后运行




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

由图可以看到模型比初始化全部为 0 效果要好一点，查看分类结果

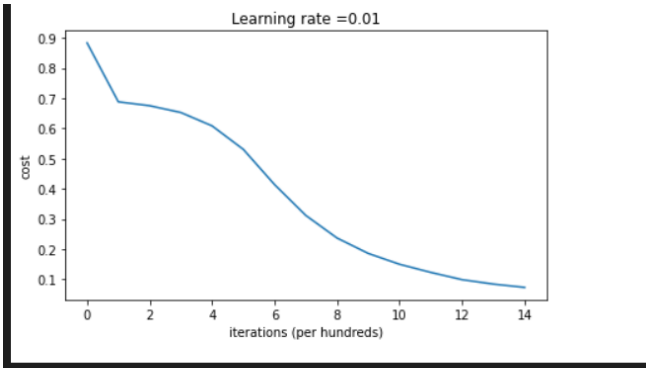


4. He initialization

使用给定公式初始化参数

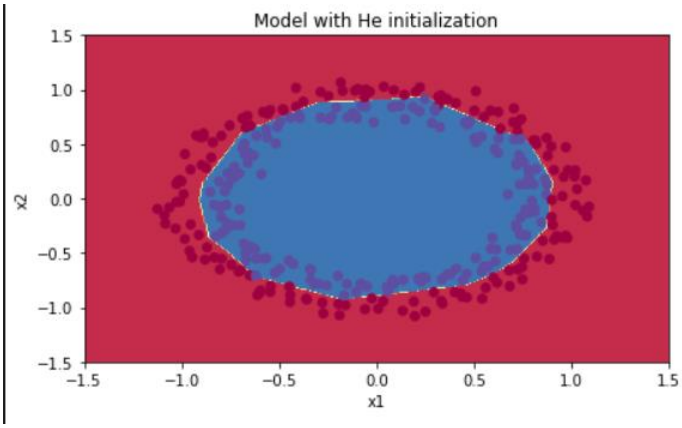
```python
for l in range(1, L + 1):
    ### START CODE HERE ### (≈ 2 lines of code)
    parameters['W' + str(l)] = np.random.randn(layers_dims[l], layers_dims[l - 1]) * np.sqrt(2 / layers_dims[l - 1])
    parameters['b' + str(l)] = np.zeros((layers_dims[l], 1))
    ### END CODE HERE ###
```

使用得到的参数进行训练预测




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

由图可以看到效果比随机设置的参数好，误差越来越小，查看分类效果



总结：不同的初始化方法可能导致性能最终不同，随机初始化有助于打破对称，使得不同隐藏层的单元可以学习到不同的参数。初始化时，初始值不宜过大。He 初始化搭配 ReLU 激活函数可以得到不错的效果

步骤二 Gradient Checking

梯度计算公式如下

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

1.1-dimensional gradient checking

由公式可以得到

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

实现反向传播：

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

进行梯度检查，梯度检查的步骤如下：

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

接下来，计算梯度的反向传播值，最后计算误差，当 difference 小于 10^{-7} 时，我们通常认为我们计算的结果是正确的

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

根据公式实现代码如下：

```
### 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
### END CODE HERE ###

# 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)                                  # Step 1'
denominator = np.linalg.norm(grad) + np.linalg.norm(gradapprox)                # Step 2'
difference = numerator / denominator                                           # Step 3'
### END CODE HERE ###
```

进行测试结果显示正确

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

✓ 0.3s

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

2. N-dimensional gradient checking

实现梯度检查的伪代码如下：

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

根据公式实现代码如下：

```
    # Compute J_plus[i]. Inputs: "parameters_values, epsilon". Output = "J_plus[i]".
    # "_" is used because the function you have to outputs two parameters but we only care about the first o
    ### 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,gc_utils.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,gc_utils.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'
```

进行测试结果显示与官方文档中的一致：

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

✓ 0.7s

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

Expected output:

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

结果说明 backward_propagation_n 代码中似乎有错，通过实现梯度检查，检查 dW2 和 db1 纠正错误，修改代码如下：

```
#dW2 = 1. / m * np.dot(dZ2, A1.T) * 2  # Should not multiply by 2
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.i  T: Any  > 0))
dW1 = 1./m * np.dot(dZ1, X.T)
#db1 = 4. / m * np.sum(dZ1, axis=1, keepdims=True) # Should not multiply by 4
```

已修复完成后，重新运行梯度检查，测试结果显示正确

```
difference = gradient_check_n(parameters, gradients, X, Y)
```

✓ 0.4s

```
Your backward propagation works perfectly fine! difference = 1.1890913024229996e-07
```

步骤三 Optimization

在深度学习中，如果数据集没有足够大的话，可能会导致一些过拟合的问题。过拟合导致的结果就是在训练集上有着很高的精确度，但是在遇到新的样本时，精确度下降会很严重。为了避免过拟合的问题，使用正则化的方式

1. Gradient Descent

梯度下降参数更新规则：

$$W^{[l]} = W^{[l]} - \alpha \, dW^{[l]}$$
$$b^{[l]} = b^{[l]} - \alpha \, db^{[l]}$$

根据公式实现代码如下：

```
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 ###
```

2. Mini-Batch Gradient descent

首先从训练集（X，Y）构建小批量，创建训练集（X，Y）的混洗版本， X 和 Y 的每一列代表一个训练示例。将混洗后的（X，Y）分成大小为“mini_batch_size”（此处为 64）的小批量，更新公式如下：

```
first_mini_batch_X = shuffled_X[:, 0 : mini_batch_size]
second_mini_batch_X = shuffled_X[:, mini_batch_size : 2 * mini_batch_size]
...
```

实现代码如下：

```
# Step 2: Partition (shuffled_X, shuffled_Y). Minus the end case.
num_complete_minibatches = math.floor(m/mini_batch_size) # number of mini batches of size mini_batc
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)

# Handl (parameter) mini_batch_size: Any < mini_batch_size)
if m % mini_batch_size != 0:
    ### START CODE HERE ### (approx. 2 lines)
    mini_batch_X = shuffled_X[:, -(m - mini_batch_size * num_complete_minibatches):]
    mini_batch_Y = shuffled_Y[:, -(m - mini_batch_size * num_complete_minibatches):]
    ### END CODE HERE ###
```

3. Momentum

小批量梯度下降只看到一个样本子集后进行参数更新，更新的方向有一定的差异，所以小批量梯度下降所采取的路径会“振荡”向收敛。 使用动量可以减少这些振荡。Momentum 考虑了过去的梯度来平滑更新。 将先前梯度的“方向”存储在变量 v 中，是指先前步骤梯度的指数加权平均值，也可以将 v 视为滚下山坡的球的“速度”，根据坡度/坡度的方向增加速度（和动量）

更新公式如下：

```
v["dW" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["W" + str(l+1)])
v["db" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["b" + str(l+1)])
```

实现代码如下：

```
# 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 ###
```

接下来使用动量实现参数更新。 动量更新规则是：

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

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

其中 L 是层数，beta 是动量，alpha 是学习率，实现代码如下：

```
# Momentum update for each parameter
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)] -= learning_rate * v["dW" + str(l+1)]
    parameters["b" + str(l+1)] -= learning_rate * v["db" + str(l+1)]
    ### END CODE HERE ###
```

速度初始化为零，算法将进行几次迭代以“建立”速度并开始采取更大的步骤。如果 beta = 0，那么这就是没有动量的标准梯度下降。动量 beta 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 beta 太大，它也会过度平滑更新。

4. Adam

Adam 结合了 RMSProp 和 Momentum 的想法，计算过去梯度的指数加权平均值，并将其存储在变量 v（在偏差校正之前）和 v^{corrected}（在偏差校正之前）。计算过去梯度平方的指数加权平均值，并将其存储在变量 s（在偏差校正之前）和 s^{corrected}（在偏差校正之前）。根据组合来自“1”和“2”的信息在一个方向上更新参数，更新规则如下：

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

t 计算 Adam 采取的步数，L 是层数，beta_1 和 beta_2 是控制两个指数加权平均值的超参数，alpha$ 是学习率，varepsilon$ 是一个非常小的数字，以避免被零除，实现代码如下：

```
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 ###
```

现在，使用 Adam 实现参数更新，一般的更新规则如下：

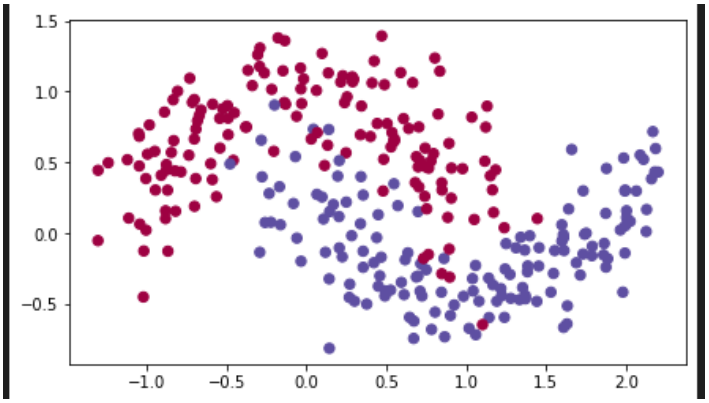
$$\begin{cases} v_{W^{[l]}} = \beta_1 v_{W^{[l]}} + (1-\beta_1)\frac{\partial J}{\partial W^{[l]}} \\ v_{W^{[l]}}^{corrected} = \frac{v_{W^{[l]}}}{1-(\beta_1)^t} \\ s_{W^{[l]}} = \beta_2 s_{W^{[l]}} + (1-\beta_2)(\frac{\partial J}{\partial W^{[l]}})^2 \\ s_{W^{[l]}}^{corrected} = \frac{s_{W^{[l]}}}{1-(\beta_2)^t}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

实现代码如下：

```
for l in range(L):
    # Moving average of the gradients. Inputs: "v, grads, beta1". Output: "v".
    ### 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_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) * (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 - 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)] -= 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))
    ### END CODE HERE ###
```

5. Model with different optimization algorithms

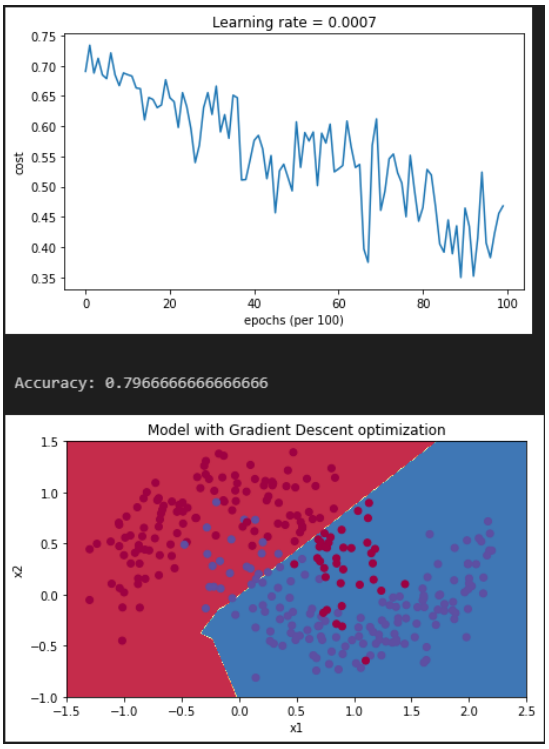
使用“moons”数据集来测试不同的优化方法，加载数据如下：



然后使用 3 种优化方法中的每一种运行 3 层神经网络

5.1 - Mini-batch Gradient descent

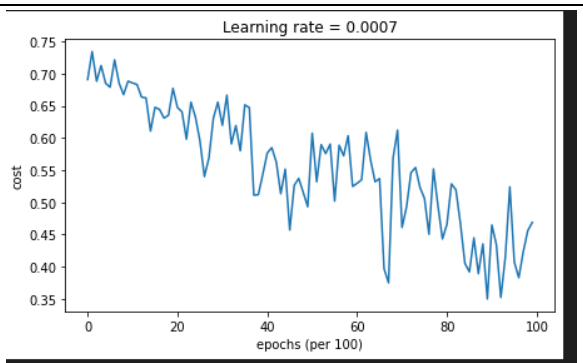
查看模型如何使用小批量梯度下降，实现效果如图所示：



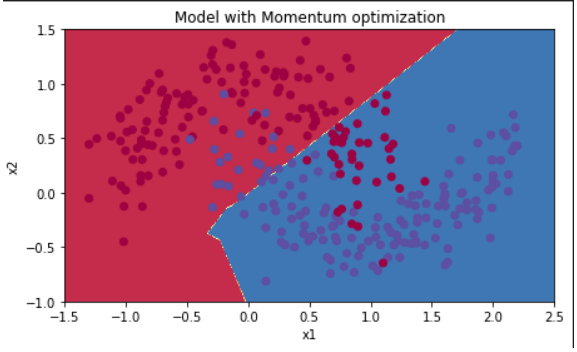
由图可知该模型准确率很高，Accuracy = 0.796,效果很好。

5.2 - Mini-batch gradient descent with momentum

查看模型如何处理动量，由于例子比较简单，使用 momentum 的收益很小，实现效果如图所示：



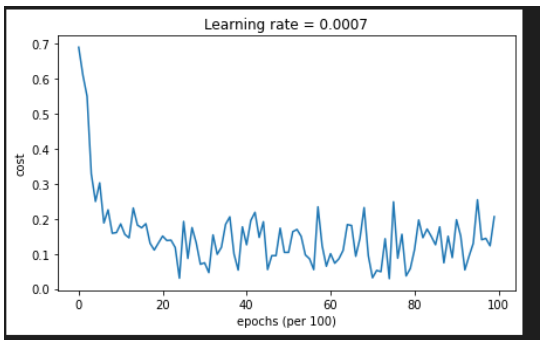

```
Accuracy: 0.7966666666666666
```



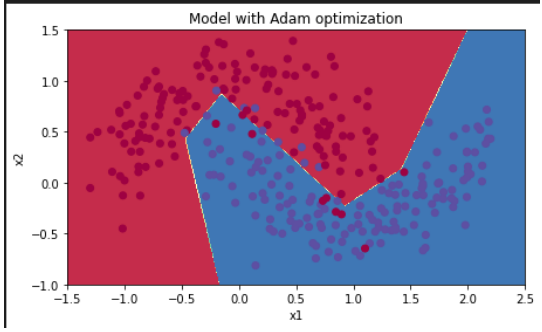
可见该模型得到的效果与梯度下降效果差不多

5.3 - Mini-batch with Adam mode

查看模型如何处理 Adam，实现效果如图所示：




```
Accuracy: 0.94
```



可见使用 Adam 模型得到模型准确率最高，Accuracy = 0.94,效果最好

结论分析与体会：

Momentum 通常有效果，但考虑到小的学习率和简单的数据集，它的影响几乎可以忽略不计。 此外，对于优化算法来说，一些小批量比其他小批量更难。另一方面，Adam 明显优于小批量梯度下降和 Momentum，Adam 收敛得更快。

Adam 的一些优点包括：

- 内存需求相对较低（虽然高于梯度下降和带有动量的梯度下降）
- 通常即使很少调整超参数也能很好地工作（除了 alpha）

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：

这次实验总体上来说就是内容比较明确，实验过程中代码大多数情况下可以由给定公式推导出来，主要的难点在于不清楚 python 中对于列表、字典的运算，花费了点时间在网上学习。其实，最难的部分还是在优化部分，虽然指导书上有讲解，但是理解起来还是有些困难。