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：Homework 2_1		学号：201900130015
日期：2021.10.7	班级： 智能班	姓名：李德锋

Email：ldf2878945468@163.com

实验目的：
掌握基本的神经网络调整技能，并尝试改进深度神经网络：超参数调整、正则化和优化

实验软件和硬件环境：
Intel(R) Core(TM) i7-8550U CPU

实验原理和方法：
利用提示的公式和原理进行填充

实验步骤：（不要求罗列完整源代码）
Initialization
1. 将所有参数初始化为零

- the weight matrices $(W^{[1]}, W^{[2]}, W^{[3]}, ..., W^{[L-1]}, W^{[L]})$
- the bias vectors $(b^{[1]}, b^{[2]}, b^{[3]}, ..., b^{[L-1]}, b^{[L]})$

Use np.zeros((.....)) with the correct shapes.

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

结论：权重应该随机初始化，以打破对称性
2. 将权重初始化为大的随机值(按*10 缩放)，并将偏差初始化为零

np.random.randn(..,..) * 10 for weights and np.zeros((.., ..)) for biases.

```
parameters['W' + str(l)] = np.random.randn(layers_dims[l], layers_dims[l-1])*10
            parameters['b' + str(l)] = np.zeros(shape=(layers_dims[l], 1))
```

结论：将权重初始化为非常大的随机值效果不好。
3. He initialization.初始化

sqrt(2./layers_dims[l-1]).)

```
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(shape=(layers_dims[l], 1))
            ### END CODE HERE ###
```

结论：He initialization works well for networks with ReLU activations.

Gradient Checking:

1. 1-dimensional gradient checking

$J(\theta) = \theta x.$

```
J = theta*x
```

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

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

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

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

2. N-dimensional gradient checking

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^{+})).

```
thetaplus = np.copy(parameters_values)                                  #
thetaplus[i][0] +=epsilon                              # Step 2
J_plus[i], _ = forward_propagation_n(X, Y, vector_to_dictionary(thetaplus))
```

发现结果不对，调整反向传播函数得到正确结果

```
Your backward propagation works perfectly fine! difference = 1.1885552035482147e-07
```

结论：Gradient checking 验证反向传播的梯度和梯度的数值之间的接近度

Optimization Methods:

1. Gradient Descent

$$W^{[l]} = W^{[l]} - \alpha \; dW^{[l]}$$

$$b^{[l]} = b^{[l]} - \alpha \; db^{[l]}$$

```
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)]
```

结论：

The difference between gradient descent, mini-batch gradient descent and stochastic gradient descent is the number of examples you use to perform one update step

调整学习速率超参数；对于良好小批量梯度下降，它通常优于梯度下降或随机梯度下降(尤其是当训练集很大时)

2. Mini-Batch Gradient descent

```
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]
```

洗牌和分区是构建小批量所需的两个步骤

通常选择 2 的幂作为小批量，例如 16、32、64、128。

3. Momentum

```
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)])
```

```
v["dW" + str(l+1)] = np.zeros(shape=parameters['W' + str(l+1)].shape)
v["db" + str(l+1)] = np.zeros(shape=parameters['b' + str(l+1)].shape)
```

结论：动量考虑了过去的梯度，以平滑梯度下降的步骤。它可以应用于批量梯度下降，小批量梯度下降或随机梯度下降

4. Adam

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

```
v["dW" + str(l+1)] = np.zeros(shape=parameters["W" + str(l+1)].shape)
v["db" + str(l+1)] = np.zeros(shape=parameters["b" + str(l+1)].shape)
s["dW" + str(l+1)] = np.zeros(shape=parameters["W" + str(l+1)].shape)
s["db" + str(l+1)] = np.zeros(shape=parameters["b" + str(l+1)].shape)
```

5.Model with different optimization algorithms
动量通常有所帮助，但是考虑到小的学习率和简单的数据集，它的影响几乎是疏忽的
Adam 明显优于小批量梯度下降和动量，相对较低的内存需求；即使对超参数调整很少，通常也能很好地工作

结论分析与体会：

Initialization：随机初始化用于打破对称性，确保不同的隐藏单元可以学习不同的东西；不要初始化太大的值 ；He initialization 对带有 ReLU activations 的网络很有效。

Gradient Checking：梯度检查很慢！近似梯度的计算成本很高。在训练期间的每次迭代中不会运行梯度检查。只需几次即可检查渐变是否正确。
梯度检查不适用于 dropout。在没有 dropout 的情况下运行梯度检查算法以确保您的反向传播正确，然后添加 dropout。

Optimization：
动量通常有所帮助，但是考虑到小的学习率和简单的数据集，它的影响几乎是疏忽的
Adam 明显优于小批量梯度下降和动量，相对较低的内存需求；即使对超参数调整很少，通常也能很好地工作

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：

问题：有些原理和公式难以理解
解决：上网搜索和与同学讨论