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：神经网络实现		学号：201600301148
日期：3.24	班级：人工智能	姓名：周阳
Email：862077860@qq.com		

实验目的：
1. 了解基本的图像分类方法。
2. 调整超参数，方法，在高层语义的基础上对深度学习网络进行调整
3. 了解基本的深度学习方法：梯度检验（数值梯度与解析梯度），初始化方法（随机，HE，零初始化等），了解优化方法，了解正则化方法

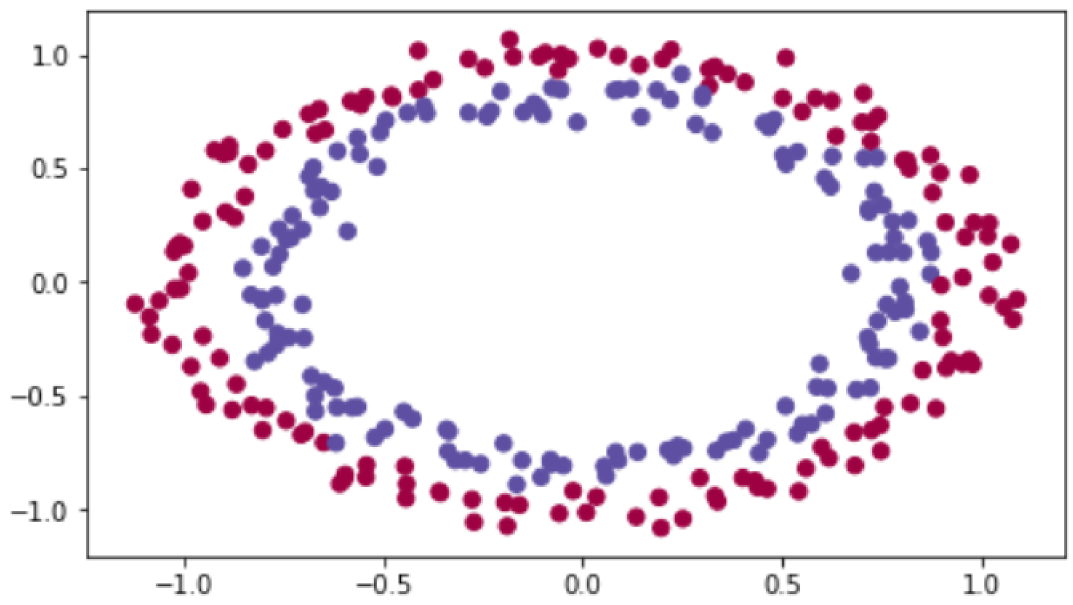
实验软件和硬件环境：

CPU：英特尔至强 E5
GPU：NVIDIA GeForce 1060 6G
内存：16G
Pycharm
Python 3.6

实验原理和方法：

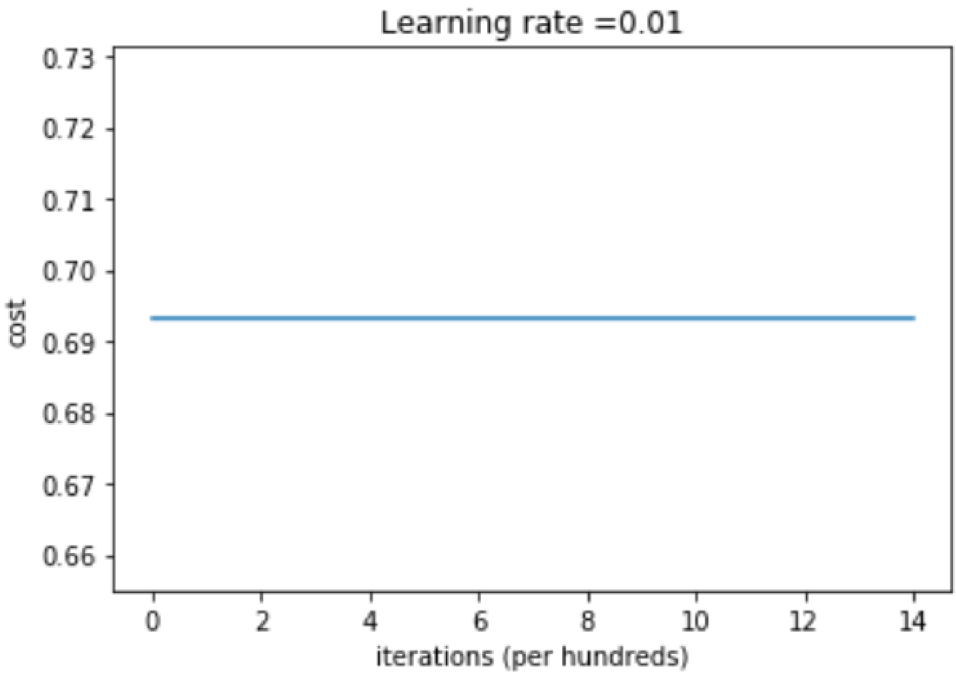
一、神经网络中的初始化，正则化，优化方法，梯度检查

①初始化
问题，二分类：

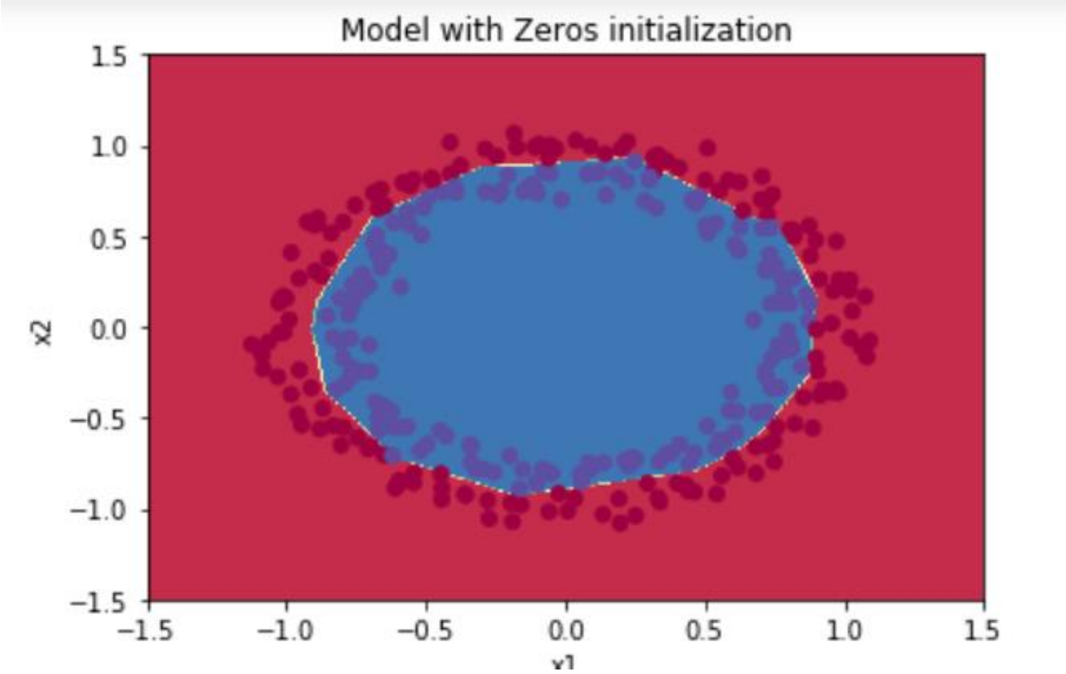


三种初始化方法：
0 初始化：使用 np.zeros()

优化过程：

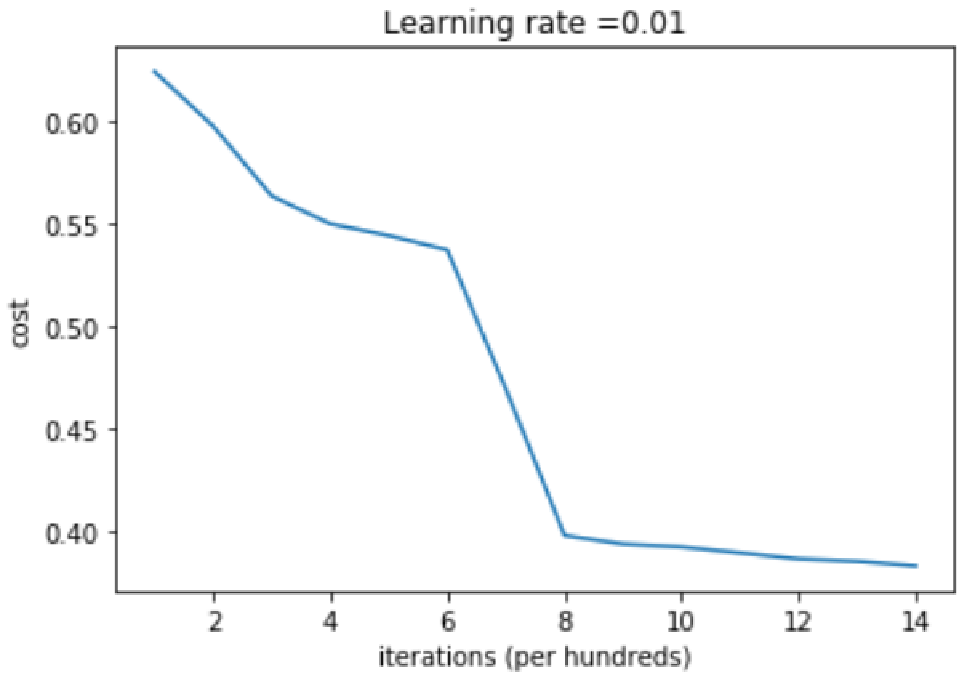


优化结果:

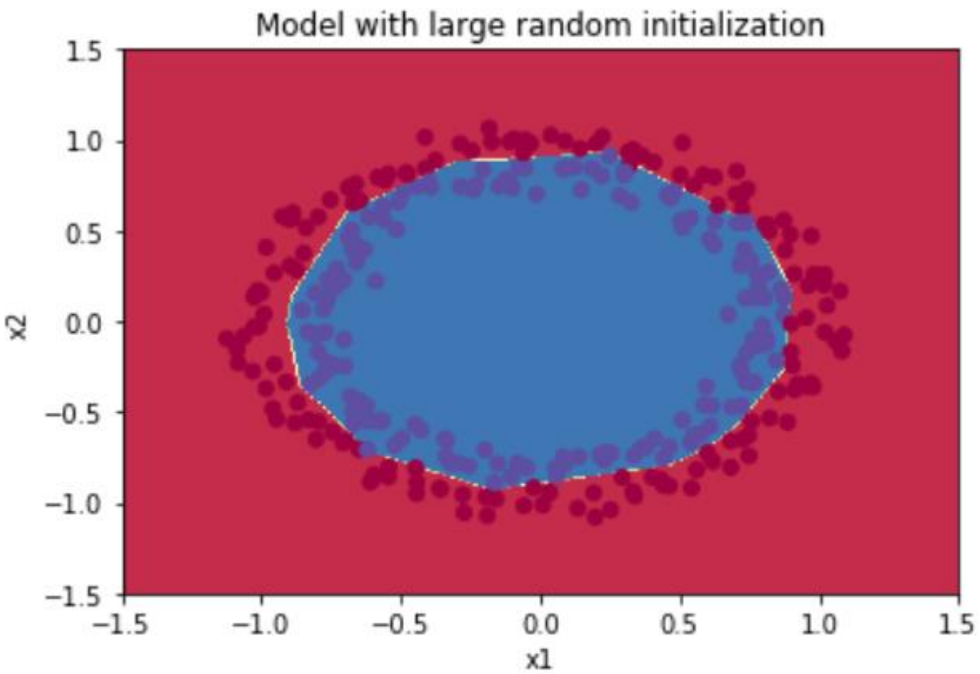


随机初始化：使用 `np.random.randn(..,..) * 10`

优化过程：

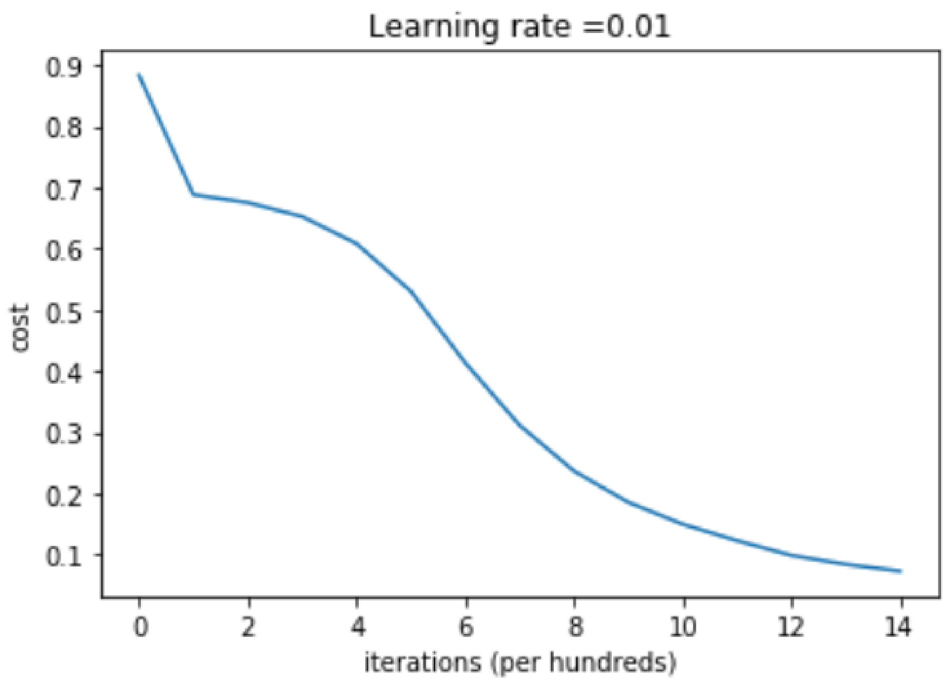


优化结果：

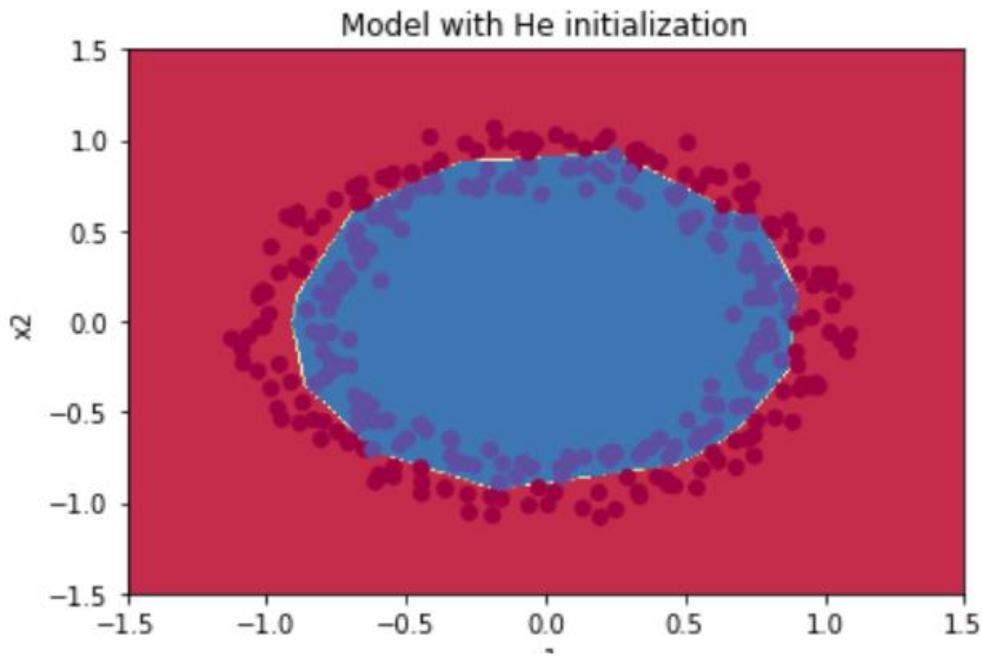


He 正则化：使用 `np.random.randn(..,..)` $* \sqrt{\frac{2}{\text{dimension of the previous layer}}}$

优化过程：



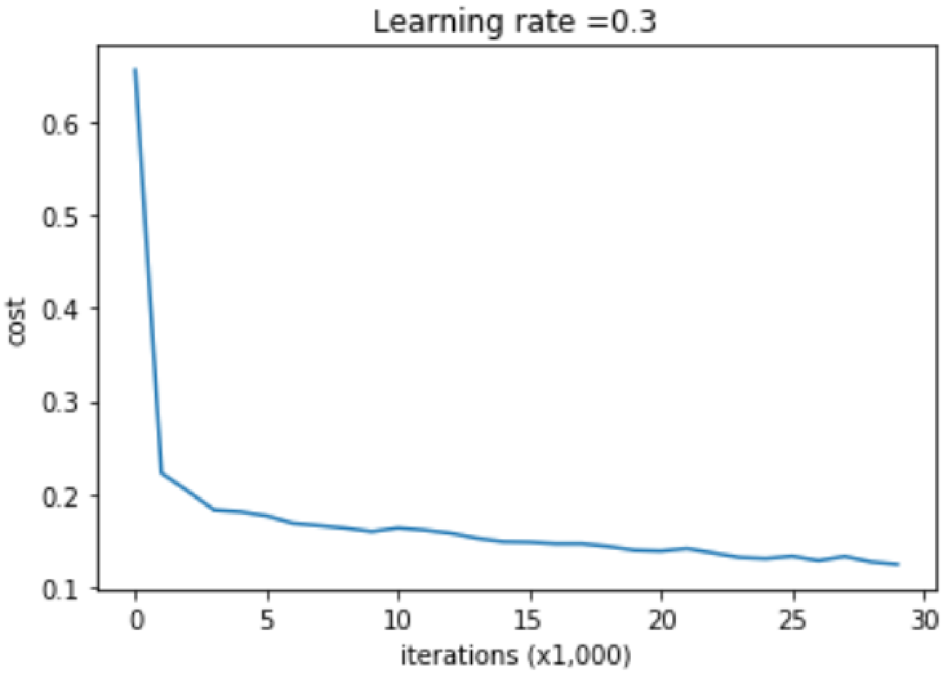
优化结果：

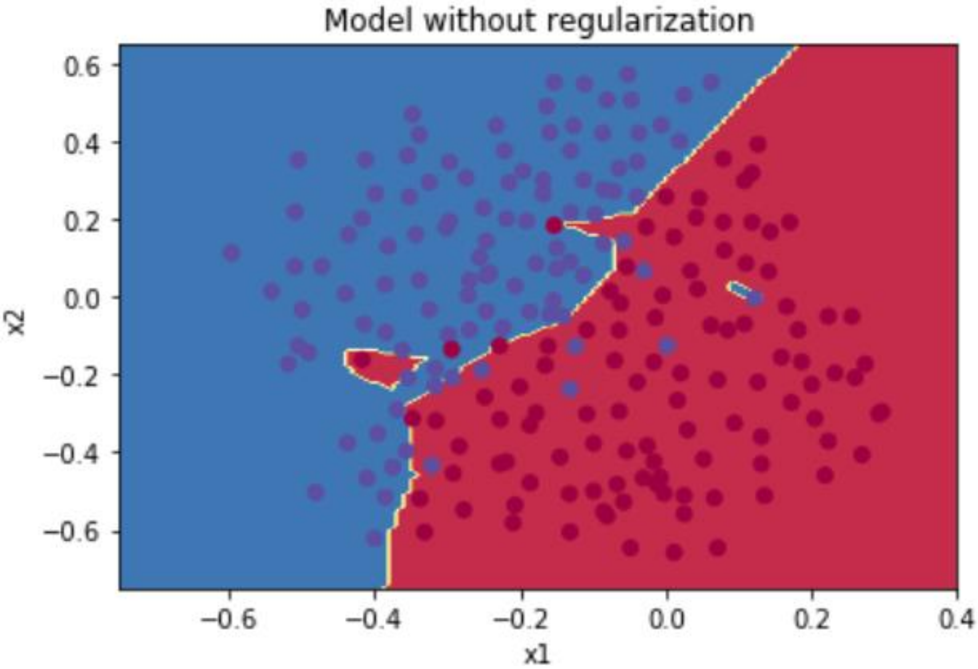


②正则化

不进行正则化：

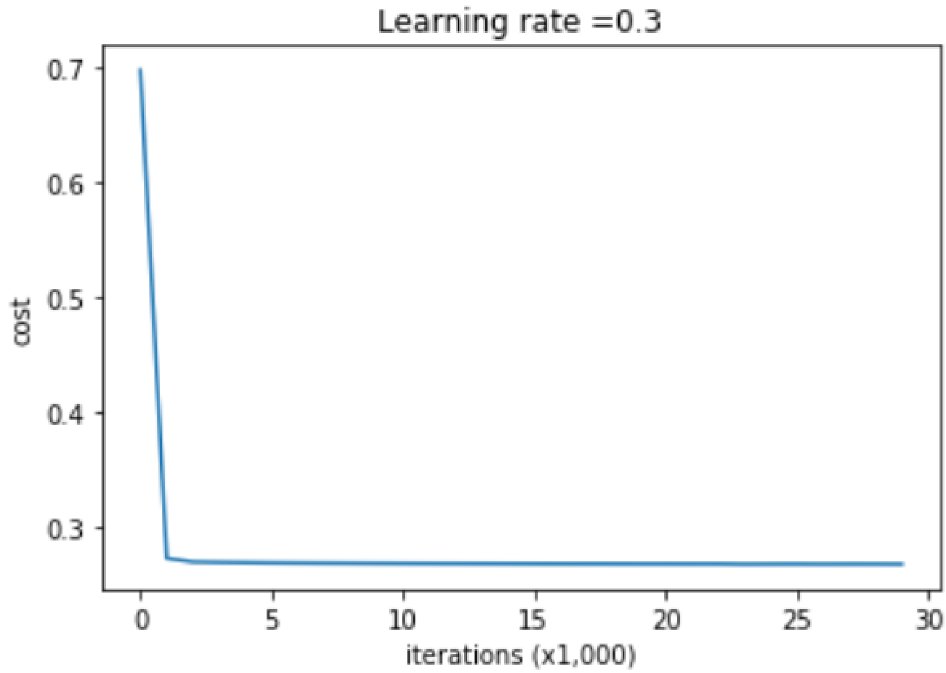
$$J = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)$$

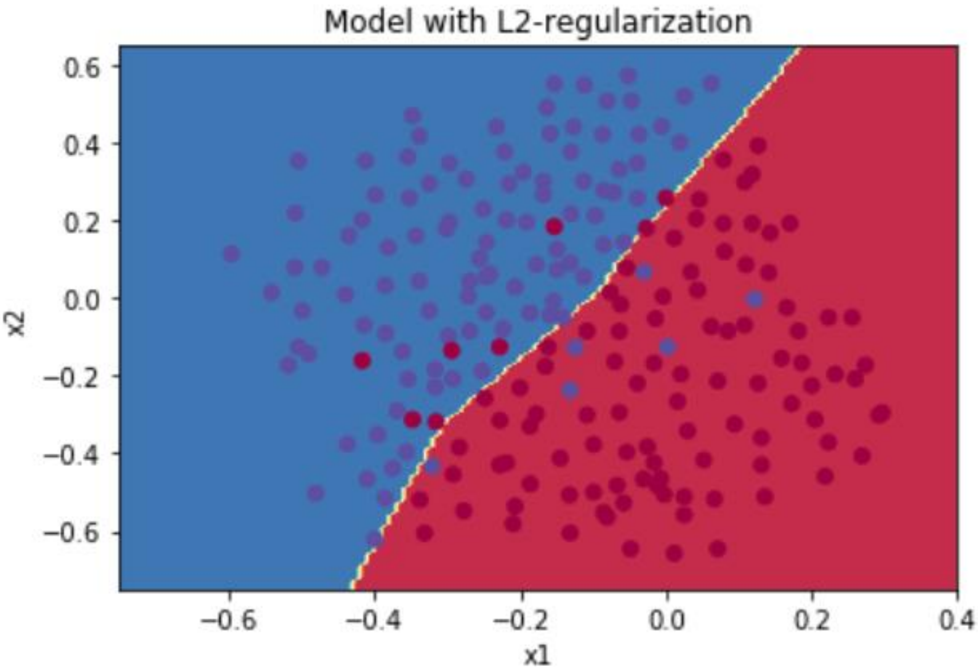




进行 L2 正则化：

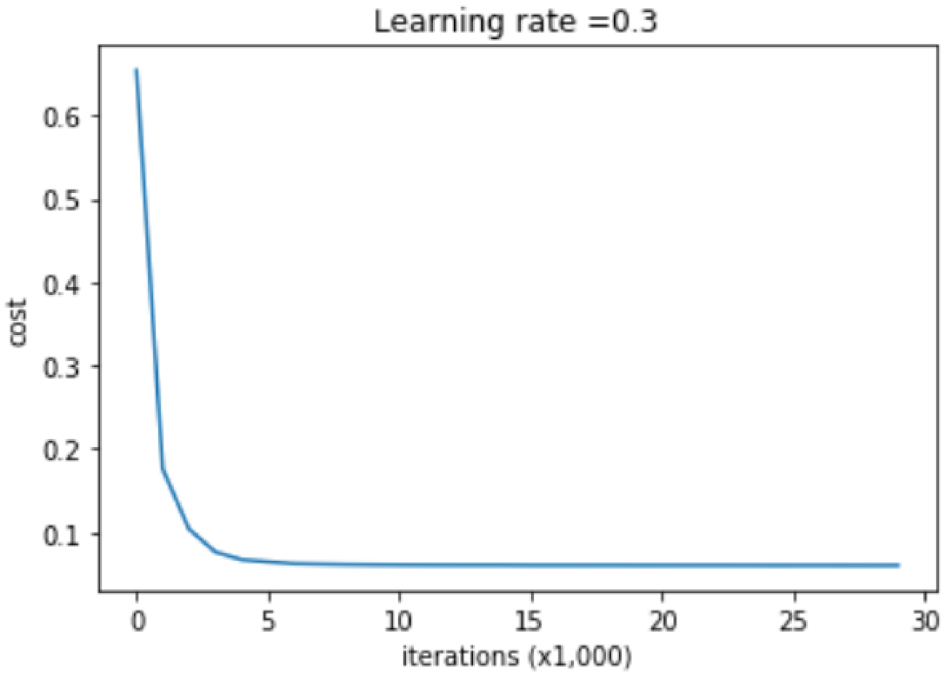
$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_l\sum_k\sum_j W_{k,j}^{[l]2}}_{\text{L2 regularization cost}}$$

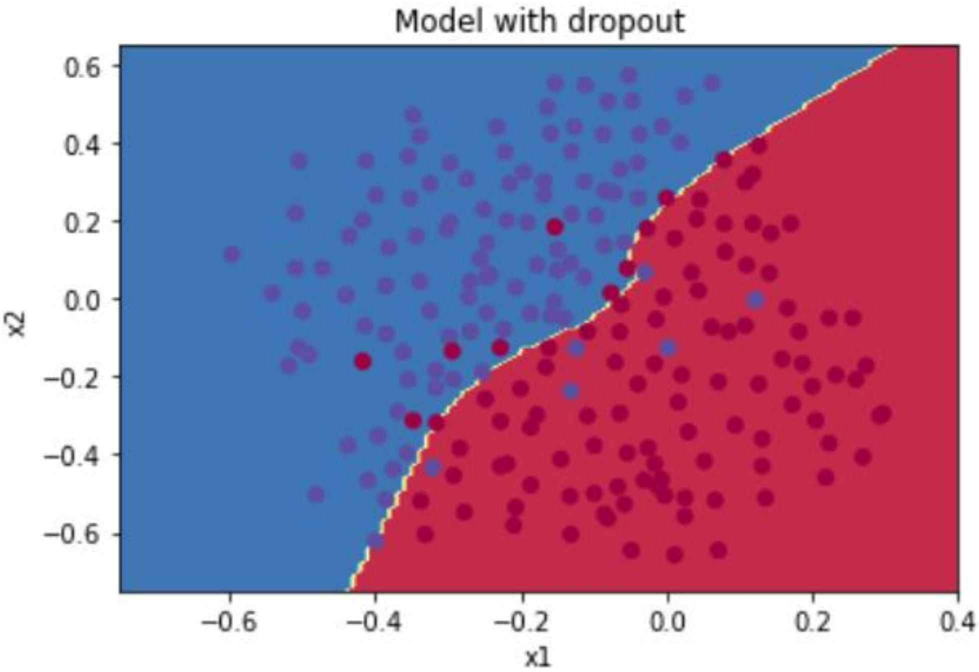




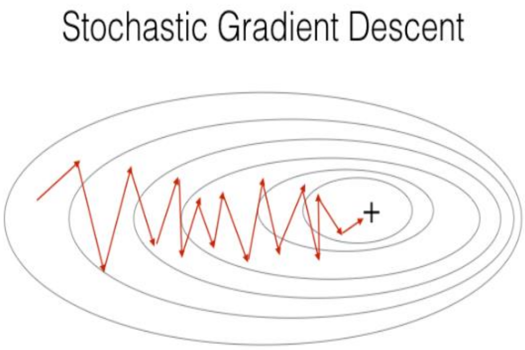
使用 Dropout：

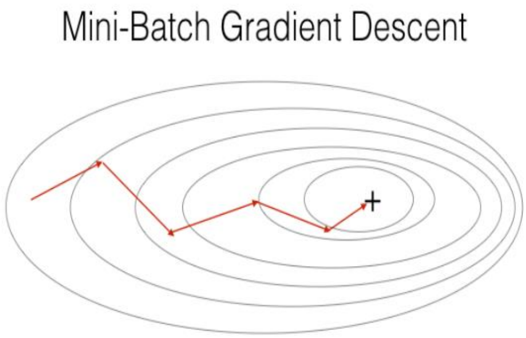
1，dropout 一般设置在激活层后面（activate function 后面）：决定一部分神经元的结果可以通过。
2，D = like a 矩阵 D 里面小于 keep_prob 的神经元可以通过。
3，使用 A * （D<k_p） 来通过
4，并进行放缩 A/k_p
5，反向传播时，从后向前使用 drop_out，使用同样的（D<k_p）* dA 并且 也放缩 dA/k_p


Learning rate =0.3
cost
iterations (x1,000)


Model with dropout
x2
x1

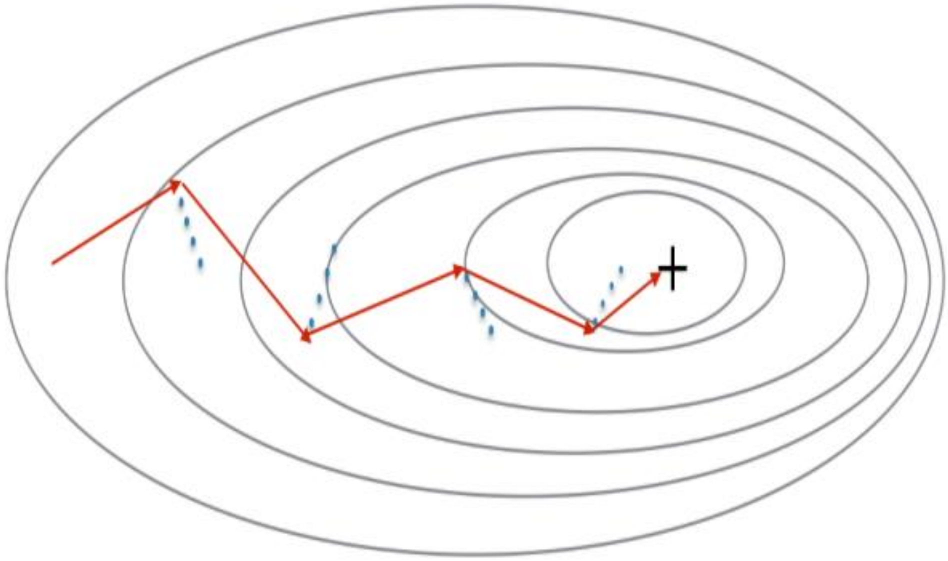
③梯度优化
梯度下降：





动量法与 Adam：

动量法：



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

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

Adam:

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

④梯度检验

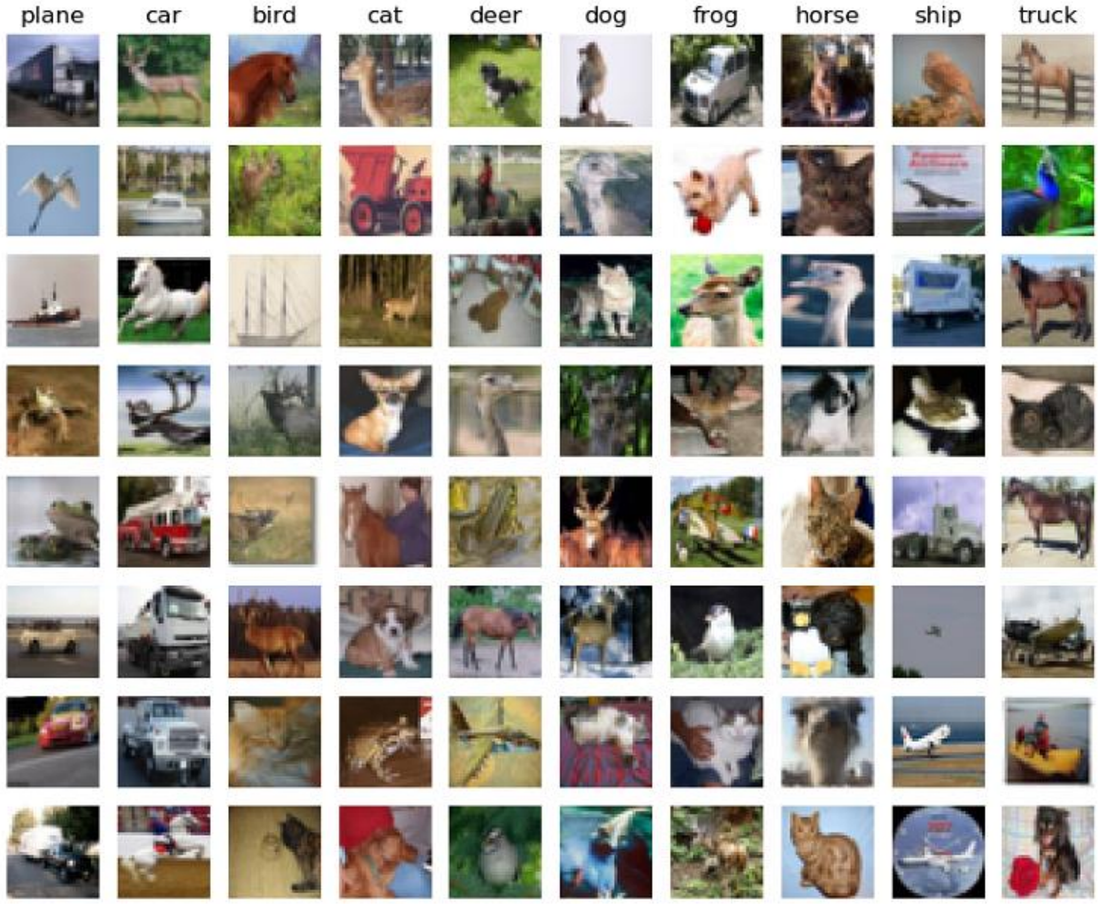
梯度检验公式（1d 与 nd）：

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

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

二、超参数调整

①数据可视化



Cifar-10 数据集

②baseline 结果

SVM valid acc: 0.413

```
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.114388 val accuracy: 0.122000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.099673 val accuracy: 0.098000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.118653 val accuracy: 0.112000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.109633 val accuracy: 0.105000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.344388 val accuracy: 0.334000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.400163 val accuracy: 0.402000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.414265 val accuracy: 0.413000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.401878 val accuracy: 0.404000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.347796 val accuracy: 0.337000
best validation accuracy achieved during cross-validation: 0.413000
```

③调参结果

超参数使用：

使用 homework1 的超参数， 得到结果如下所示：

```
hidden_size = 50
num_classes = 10
batch_size = 128
lr = 5e-3
reg = 0.01
```

num_batch = 50000

learning_rate_decay = 0.99

使用 learning rate 缩减

```
iteration 49200 / 50000: loss 2.302956  train_acc 0.078125
iteration 49300 / 50000: loss 2.302078  train_acc 0.078125
iteration 49400 / 50000: loss 2.302719  train_acc 0.078125
iteration 49500 / 50000: loss 2.302192  train_acc 0.109375
iteration 49600 / 50000: loss 2.302904  train_acc 0.046875
iteration 49700 / 50000: loss 2.302650  train_acc 0.031250
iteration 49800 / 50000: loss 2.302673  train_acc 0.062500
iteration 49900 / 50000: loss 2.302642  train_acc 0.101562
0.1
```

Valid 准确率：0.1

经过调整超参数得到最佳的超参数为：

```
best hyp{'std': 1, 'lr': 0.01, 'batch_size': 2048, 'hiddden_size': 256, 'valid': 0.552}
```

使用 he 初始化，实验中发现 std（初始化系数）对结果的影响巨大。

最佳 valid 准确率：0.552 其在 test 上的准确率为 0.55

实验步骤：（不要求罗列完整源代码）

一、神经网络基本方法

初始化：

```
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 ###
```

```
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 ###
```

```
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))
```

正则化:

```
L2_regularization_cost = lambd*(np.sum(np.square(W1)) + np.sum(np.square(W2)) + np.sum(np.square(W3)))/m/2
```

```
### START CODE HERE ### (approx. 1 line)
dW3 = 1./m * np.dot(dZ3, A2.T) + lambd/m * W3
### END CODE HERE ###
db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)

dA2 = np.dot(W3.T, dZ3)
dZ2 = np.multiply(dA2, np.int64(A2 > 0))
### START CODE HERE ### (approx. 1 line)
dW2 = 1./m * np.dot(dZ2, A1.T) + lambd/m * W2
### END CODE HERE ###
db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)

dA1 = np.dot(W2.T, dZ2)
dZ1 = np.multiply(dA1, np.int64(A1 > 0))
### START CODE HERE ### (approx. 1 line)
dW1 = 1./m * np.dot(dZ1, X.T) + lambd/m * W1
### END CODE HERE ###
```

```
### START CODE HERE ### (approx. 4 lines)          # Steps 1-4 below correspond to the Steps 1-4 described above.
D1 = np.random.rand(A1.shape[0],A1.shape[1])                                    # Step 1: initialize matrix D1 = np.random.rand(..., ..
D1 = D1 <= keep_prob                                     # Step 2: convert entries of D1 to 0 or 1 (using keep_prob as the threshold)
A1 = A1 * D1                                       # Step 3: shut down some neurons of A1
A1 = A1 / keep_prob                                      # Step 4: scale the value of neurons that haven't been shut down
### END CODE HERE ###
Z2 = np.dot(W2, A1) + b2
A2 = relu(Z2)
### START CODE HERE ### (approx. 4 lines)
D2 = np.random.rand(A2.shape[0],A2.shape[1])                                    # Step 1: initialize matrix D2 = np.random.rand(..., .
D2 = D2 <= keep_prob                                     # Step 2: convert entries of D2 to 0 or 1 (using keep_prob as the threshold)
A2 = A2 * D2                                          # Step 3: shut down some neurons of A2
A2 = A2/keep_prob                                       # Step 4: scale the value of neurons that haven't been shut down
### END CODE HERE ###
Z3 = np.dot(W3, A2) + b3
A3 = sigmoid(Z3)
```

```
### START CODE HERE ### (≈ 2 lines of code)
dA2 = dA2 * D2              # Step 1: Apply mask D2 to shut down the same neurons as during the forward propagation
dA2 = dA2 / keep_prob              # Step 2: Scale the value of neurons that haven't been shut down
### END CODE HERE ###
dZ2 = np.multiply(dA2, np.int64(A2 > 0))
dW2 = 1./m * np.dot(dZ2, A1.T)
db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)

dA1 = np.dot(W2.T, dZ2)
### START CODE HERE ### (≈ 2 lines of code)
dA1 = dA1 * D1              # Step 1: Apply mask D1 to shut down the same neurons as during the forward propagation
dA1 = dA1 / keep_prob              # Step 2: Scale the value of neurons that haven't been shut down
### END CODE HERE ###
```

优化方法：

```
    ### 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)

# Handling the end case (last mini-batch < mini_batch_size)
if m % mini_batch_size != 0:
    ### START CODE HERE ### (approx. 2 lines)
    mini_batch_X = shuffled_X[:, -(m%mini_batch_size) : ]
    mini_batch_Y = shuffled_Y[:, -(m%mini_batch_size) : ]
    ### END CODE HERE ###
```

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

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

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

```
### 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-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_corrected".
### 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 ###

# 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 ###
```

梯度检验：

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

```
        ### 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, 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, 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'
    ### END CODE HERE ###
```

二、超参数调整

超参数部分代码：

```
stds = [1]
lrs = [1e-2,1e-3,1e-1]
hidden_sizes = [256,512]
batch_sizes = [1024,2048]

nets = []
valids = []
hyps = []
for std in stds:
    for lr in lrs:
        for batch_size in batch_sizes:
            for hidden_size in hidden_sizes:
                net = ThreeLayerNet(input_size, hidden_size, num_classes, std=std)
                stats = net.train(X_train_feats, y_train, X_val_feats, y_val,
                                  num_iters=num_batch, batch_size=batch_size,
                                  learning_rate=lr, learning_rate_decay=learning_rate_decay,
                                  reg=reg, verbose=True)
```

```
                valid = (net.predict(X_test_feats) == y_test).mean()
                nets.append(net)
                valids.append(valid)

hyps.append({"std":std,"lr":lr,"batch_size":batch_size,"hiddden_size":hidden_size,"valid":valid})
                print("------------------std : " +str(std) +" lr: "+str(lr) +" batch_size: " + str(batch_size) + " hidden_size: " + str(hidden_size) +" valid: "+str(valid)+" --------------------")
```

使用 he initialize:

```
self.params = {}
self.params['W1'] = std * np.random.randn(input_size,hidden_size) * np.sqrt(2./hidden_size)
self.params['b1'] = np.zeros(hidden_size)
self.params['W2'] = std * np.random.randn(hidden_size, hidden_size) * np.sqrt(2./hidden_size)
self.params['b2'] = np.zeros(hidden_size)
self.params['W3'] = std * np.random.randn(hidden_size, output_size) * np.sqrt(2./output_size)
self.params['b3'] = np.zeros(output_size)
```

结论分析与体会：

1，深度学习神经网络里面有非常多的方法（trick）值得我们去留意，一个好的方法可以让训练模型变得更加高效，得到的模型效果更好。

2，通过第二实验对比第一次实验发现输入原始的 feature 和使用 HOG feature 并没有特别大的提升，这也与深度学习里面自带的 representation 学习十分相关。

3，对于调参而言，本次实验发现 std（参数初始化时的缩放倍数），和 learningrate 对学习的效率影响极大。

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，调参中遇到的困难？

答：本次调参，影响比较大的两个参数一个是 learning rate 这个很好找到，并且也相对好调。但是本次调参过程中最难调整的是 std 参数，这个参数对结果影响巨大，但是一开始并不在调整的参数范围内，最终修改了模型的入口，才调整成功。（std 非常小的话，模型正确率会一直在 0.1）

2，实验中比较重要的地方（之后能用到的）？

答：①正则化（或 drop out）的理论基础。②优化方法理论（ADAM 与动量）。③初始化中的 std 与 HE 初始化方法。