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：神经网络的正则化和批归一化		学号：202020130190
日期：2022.10.10	班级： 智能	姓名：刘绪波
Email：2842353032@qq.com		

实验目的：

掌握基本的神经网络的调参技能，并学习如何对神经网络进行改进，如超参数调整，正则化，批量标准化等,本次实验主要完成两个子任务即正则化和批归一化；

实验软件和硬件环境：

软件: Dataspell 2022.2.2 ；

硬件：CPU：AMD Ryzen 7 4800U; 显卡：AMD Radeon

实验原理和方法：

1. 正则化(Regulation)：

正则化，即在成本函数中加入一个正则化项(惩罚项)，惩罚模型的复杂度，防止网络过拟合；本实验中采用 l2 正则化和 dropout 正则化：

① 对于 L2 正则化：

原理介绍：L2 正则化分为两部分，一部分是在求损失函数的时候，在损失函数后面加一个惩罚项，另一部分是在 backward prop 求偏导时加上惩罚项；

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

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

上面这个图片就展示了从没有正则化到有正则化的公式上的变化；

对于 backward prop 中我们只需要考虑到 dW1,dw2,dw3 即可：

$$\left(\frac{d}{dW}\left(\frac{1}{2}\frac{\lambda}{m}W^2\right)=\frac{\lambda}{m}W\right)$$

② 对于 dropout 正则化：

dropout（随机失活）也是一种非常使用的正则化方法；下面介绍 dropout 的几个步骤：

1. In lecture, we dicussed creating a variable $d^{[1]}$ with the same shape as $a^{[1]}$ using `np.random.rand()` to randomly get numbers between 0 and 1. Here, you will use a vectorized implementation, so create a random matrix $D^{[1]} = [d^{[1](1)}d^{[1](2)}...d^{[1](m)}]$ of the same dimension as $A^{[1]}$.
2. Set each entry of $D^{[1]}$ to be 0 with probability (`1-keep_prob`) or 1 with probability (`keep_prob`), by thresholding values in $D^{[1]}$ appropriately. Hint: to set all the entries of a matrix X to 0 (if entry is less than 0.5) or 1 (if entry is more than 0.5) you would do: `X = (X < 0.5)`. Note that 0 and 1 are respectively equivalent to False and True.
3. Set $A^{[1]}$ to $A^{[1]} * D^{[1]}$. (You are shutting down some neurons). You can think of $D^{[1]}$ as a mask, so that when it is multiplied with another matrix, it shuts down some of the values.
4. Divide $A^{[1]}$ by `keep_prob`. By doing this you are assuring that the result of the cost will still have the same expected value as without drop-out. (This technique is also called inverted dropout.)

2. 批归一化(Batch Normalization)：

BN 就是在激活函数接收输入之前对数据分布进行规范化，具体计算就是去均值归一化，将数据的分布都规范到标准正态分布中，使得激活函数的输入值落在函数较为敏感的区域，也即梯度较大的区域，从而避免梯度消失、减少训练时间。因此，BN 也通常需要放在激活函数之前。简要来说，我们对输入（N x D）求均值（D）方差（D），然后用均值和方差去归一化我们的整个输入，得到一个输出（N x D）并且我们会保存一个 running_mean 和 running_var，这两个是历史均值/方差目前均值、方差的加权和，这二者只会在训练时计算，测试时直接使用这两个数据对输入数据进行归一化。

实验步骤：（不要求罗列完整源代码）

1. 正则化(Regulation)：

主要步骤时先通过实现没有任何正则化过的模型，通过该模型对原数据集进行边界划分，得出结果，然后再实现两种正则化的方法：L2 正则化和 dropout 正则化；随后进行结果比较，观察正则化的必要性和起到的效果；

① 首先使用没有经过正则化的模型进行边界划分：此处模型已经给出，并且对每一万次的迭代都输出一个 cost；最后画出决策边界；

② 将模型进行 L2 正则化，关于如何进行 L2 正则化在实验原理部分已经介绍；下面列出 L2 正则化的模型公式和部分关键代码：

```
W2 = parameters["W2"]
W3 = parameters["W3"]

cross_entropy_cost = compute_cost(A3, Y) # This gives you the cross-entropy part of the cost

### START CODE HERE ### (approx. 1 line)
L2_regularization_cost = lambd * (np.sum(np.square(W1)) + np.sum(np.square(W2)) + np.sum(np.square(W3))) / (2 * m)
### END CODER HERE ###

cost = cross_entropy_cost + L2_regularization_cost
```

```
### START CODE HERE ### (approx. 1 line)
dW3 = 1./m * np.dot(dZ3, A2.T) + (lambd * W3) / m
### 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 * W2) / m
### 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 * W1) / m
### END CODE HERE ###
db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)

gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3,"dA2": dA2,
             "dZ2": dZ2, "dW2": dW2, "db2": db2, "dA1": dA1,
             "dZ1": dZ1, "dW1": dW1, "db1": db1}
```

③ 将模型进行 dropout 正则化，原理部分有介绍四个步骤，列出 dropout 正则化的部分实现代码：

```
    # LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID
    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)
    ### 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 = np.where(D1 <= keep_prob, 1, 0)                                    # 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 = np.where(D2 <= keep_prob, 1, 0)                                    # 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)

    cache = (Z1, D1, A1, W1, b1, Z2, D2, A2, W2, b2, Z3, A3, W3, b3)

    return A3, cache
```

```
dZ3 = A3 - Y
dW3 = 1./m * np.dot(dZ3, A2.T)
db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)
dA2 = np.dot(W3.T, dZ3)
### 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 ###
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)

gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3,"dA2": dA2,
             "dZ2": dZ2, "dW2": dW2, "db2": db2, "dA1": dA1,
             "dZ1": dZ1, "dW1": dW1, "db1": db1}

return gradients
```

2. 批量标准化(Batch Normalization)：

进行 BN 的算法步骤：求均值。求方差。对数据进行标准化（将数据规范到标准正态分布）。训练参数 γ 和 β。通过线性变换输出。

下面看具体实验中的步骤及结果展示：

1. 进行 BN：代码展示：

```
    ############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    sample_mean = np.mean(x, axis=0)
    sample_var = np.var(x, axis=0)
    x_norm = (x - sample_mean) / np.sqrt(sample_var + eps)
    out = x_norm * gamma + beta
    cache = (x, x_norm, gamma, sample_mean, sample_var, eps)

    running_mean = momentum * running_mean + (1 - momentum) * sample_mean
    running_var = momentum * running_var + (1 - momentum) * sample_var

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
elif mode == 'test':
    ##############################################################################
    # TODO: Implement the test-time forward pass for batch normalization.        #
    # Use the running mean and variance to normalize the incoming data,          #
    # then scale and shift the normalized data using gamma and beta.             #
    # Store the result in the out variable.                                      #
    ##############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    x_norm = (x - running_mean) / (np.sqrt(running_var + eps))
    out = gamma * x_norm + beta

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##############################################################################
    #                           END OF YOUR CODE                                 #
    ##############################################################################
else:
    raise ValueError('Invalid forward batchnorm mode "%s"' % mode)
```

下面进行对归一化的结果进行验证：

```
Before batch normalization:
  means:  [ -2.3814598  -13.18038246   1.91780462]
  stds:   [27.18502186 34.21455511 37.68611762]

After batch normalization (gamma=1, beta=0)
  means:  [ 1.77635684e-17  6.10622664e-17 -1.86656246e-17]
  stds:   [0.99999999 1.         1.        ]

After batch normalization (gamma= [1. 2. 3.] , beta= [11. 12. 13.] )
  means:  [11. 12. 13.]
```

```
After batch normalization (test-time):
  means:  [-0.03927354 -0.04349152 -0.10452688]
  stds:   [1.01531428 1.01238373 0.97819988]
```

test_time

经过了归一化，样本的均值变为 0，方差变为 1；结果很好；

2. 进行 backward:

代码展示：batchnorm_backward：

```
# TODO: Implement the backward pass for batch normalization. Store the    #
# results in the dx, dgamma, and dbeta variables.                          #
# Referencing the original paper (https://arxiv.org/abs/1502.03167)        #
# might prove to be helpful.                                               #
############################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

x, x_norm, gamma, sample_mean, sample_var, eps = cache
dbeta = np.sum(dout, axis=0)
dgamma = np.sum(x_norm * dout, axis=0)

N = x.shape[0]
dx_norm = dout * gamma
dx_sub_mean_1 = dx_norm / (np.sqrt(sample_var + eps))
d1_sqrt_v = np.sum(dx_norm * (x - sample_mean), axis=0)
dsqrt_v = - d1_sqrt_v / (sample_var + eps)
dv = 0.5 * dsqrt_v / np.sqrt(sample_var + eps)
dx_square = np.ones_like(x) / N * dv
dx_sub_mean_2 = dx_square * (x - sample_mean) * 2
dx_sub_mean = dx_sub_mean_1 + dx_sub_mean_2
dx = dx_sub_mean + np.ones_like(x) / N * np.sum(-dx_sub_mean, axis=0)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
############################################################################
```

结果展示：

```
dx error:  1.7029235612572515e-09
dgamma error:  7.420414216247087e-13
dbeta error:  2.8795057655839487e-12
```

代码展示：batchnorm_backward_alt：

```
# TODO: Implement the backward pass for batch normalization. Store the    #
# results in the dx, dgamma, and dbeta variables.                          #
#                                                                          #
# After computing the gradient with respect to the centered inputs, you    #
# should be able to compute gradients with respect to the inputs in a      #
# single statement; our implementation fits on a single 80-character line.#
############################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

x, x_norm, gamma, sample_mean, sample_var, eps = cache
N = x.shape[0]
dbeta = np.sum(dout, axis=0)
dgamma = np.sum(x_norm * dout, axis=0)

dx_norm = dout * gamma
dv = ((x - sample_mean) * -0.5 * (sample_var + eps)**-1.5 * dx_norm).sum(axis=0)
dm = (dx_norm * -1 * (sample_var + eps)**-0.5).sum(axis=0) + (dv * (x - sample_mean) * -2 / N).sum(axi
dx = dx_norm / (sample_var + eps)**0.5 + dv * 2 * (x - sample_mean) / N + dm / N


# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
############################################################################
```

结果展示：

```
dx difference:  1.0733384330935792e-12
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 2.07x
```

结论分析与体会：

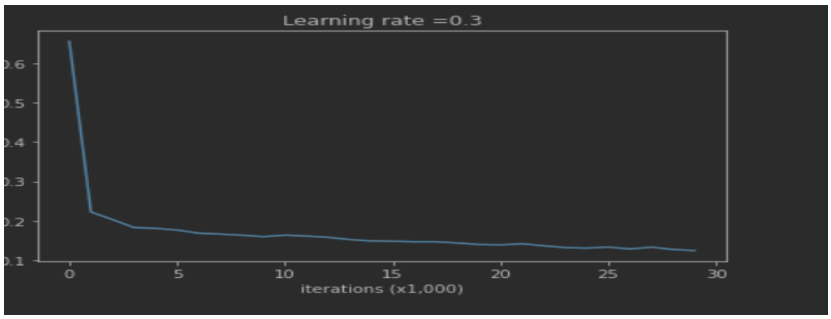
1. 正则化(Regulation)：

   结果展示：

   ① 未进行正则化的模型输出：

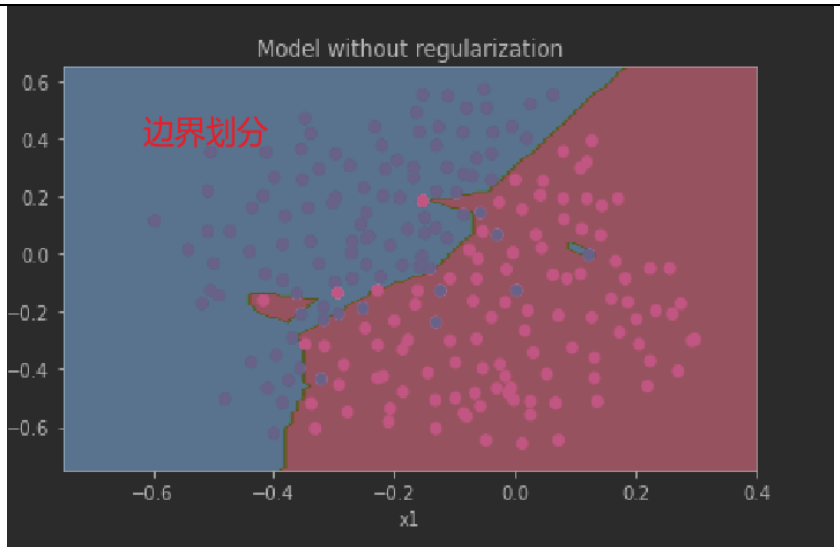
```
Cost after iteration 0: 0.6557412523481002
Cost after iteration 10000: 0.16329987525724216
Cost after iteration 20000: 0.13851642423268143
```

cost



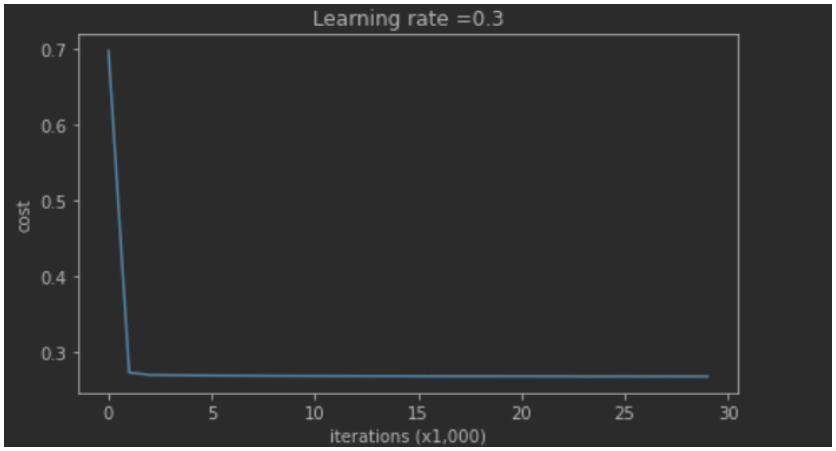

```
On the training set:
Accuracy: 0.9478672985781991
On the test set:
Accuracy: 0.915
```

Accuracy

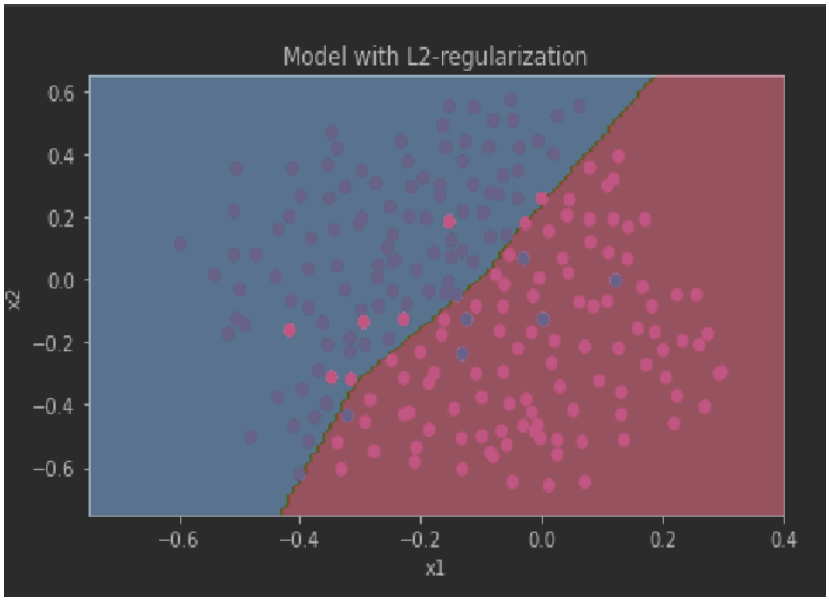


② 经过 L2 正则化的模型输出：

```
Cost after iteration 0: 0.6974484493131264
Cost after iteration 10000: 0.2684918873282238
Cost after iteration 20000: 0.2680916337127301
```

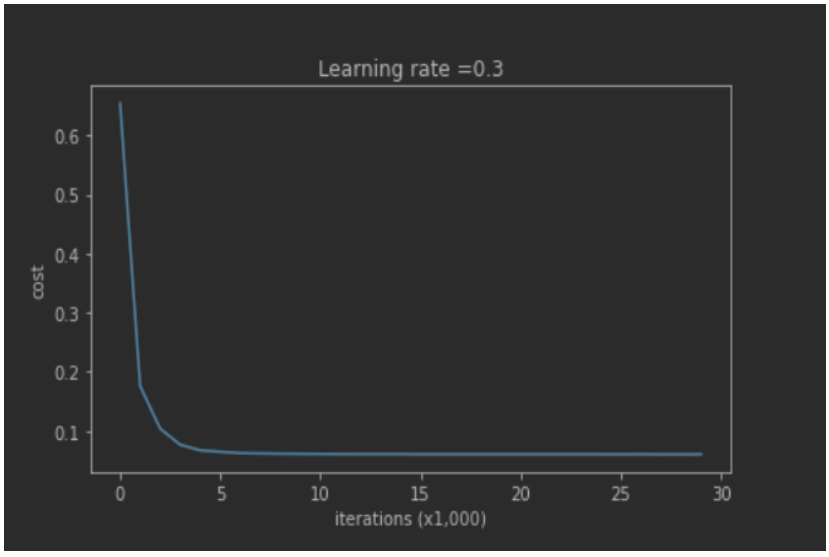



```
On the train set:
Accuracy: 0.9383886255924171
On the test set:
Accuracy: 0.93
```

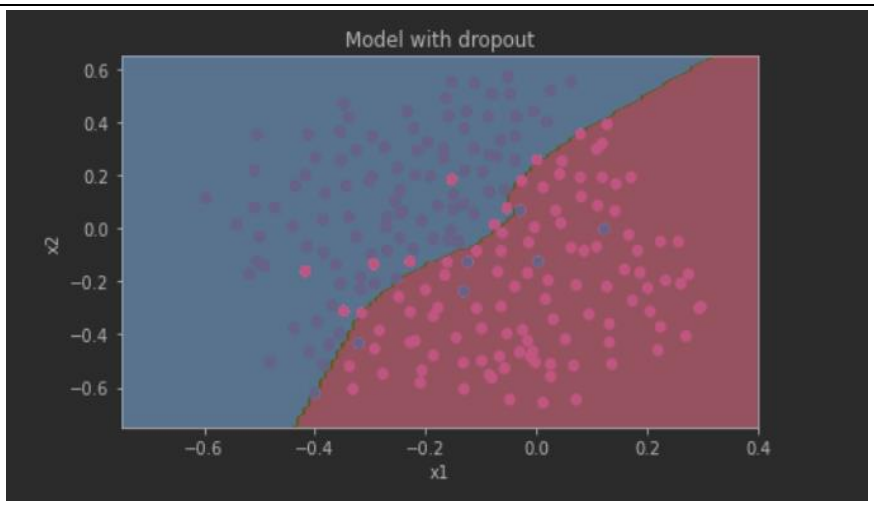


③ 经过 dropout 正则化的模型输出：

```
Cost after iteration 10000: 0.0610169865749056
Cost after iteration 20000: 0.060582435798513114
```




```
On the train set:
Accuracy: 0.9289099526066351
On the test set:
Accuracy: 0.95
```



由上面可以看出原模型出现了严重的过拟合情况，L2 正则化则对该情况进行了一定程度的避免，而 dropout 效果最好；另外，之所以 L2 正则化能够实现减轻过拟合的情况：如果正则化 λ 设置得足够大，权重矩阵 W 被设置为接近于 0 的值，直观理解就是把多隐藏单元的权重设为 0，于是基本上消除了这些隐藏单元的许多影响。如果是这种情况，这个被大大简化了的神经网络会变成一个很小的网络，小到如同一个逻辑回归单元，这个神经网络就变成了很简单的网络了，所以就避免了过拟合，当然有时候反而会导致了欠拟合。

2. 批归一化(Batch Normalization)：
   ① 对于 BN 的设计以及反向传播的结果在上面步骤中已经展现出来；
   ② 对于 BN 的作用的分析：对数据进行规范化，降低样本之间的差异。使激活函数的输入落在梯度较大的区域，一个很小的输入差异也会带来较大的梯度差异，可以有效的避免梯度消失，加快网络的收敛。降低了层与层之间的依赖关系，不加 BN 的时候当前层会直接接收上一层的输出，而加了 BN 之后当前层接收的是一些规范化的数据，因此使得模型参数更容易训练，同时降低了层与层之间的依赖关系。
   ③ 在训练阶段可以设置更高的学习率，因为 BN 有快速收敛的特性。我们在网络中加入 Batch Normalization 时，可以采用初始化很大的学习率，然后学习率衰减速度也很大，因此这个算法收敛很快。
3. 对于二者的区别：
   ① BN 是拉平不同特征图（也可以说是特征）之间的差异，进行去均值归一化。而 L2 参数权重正则化则没有改变同一层参数的相对大小，而是对当前参数自身进行正则化。
   ② 对于一个特征图，它们量级、方差可能都不同，而 BN 就让方差为 1，均值为 0，从而导致特征图之间的差异减小了。L2 参数权重正则化不以此层面为目标产生影响。
   ③ L2 参数权重正则化是对各个参数权重本身的，而 BN 是对特征图全局的。
   ④ L2 参数权重正则化让网络的权重不会过大，而 BN 的主要目的是加快训练同时防止梯度消失。

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. 在进行 BN 是出现了报错问题：

```
print_mean_std(a_norm,axis=0)

gamma = np.asarray([1.0, 2.0, 3.0])
beta = np.asarray([11.0, 12.0, 13.0])
# Now means should be close to beta and stds close to gamma
print('After batch normalization (gamma=', gamma, ', beta=', beta, ')')
a_norm, _ = batchnorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,axis=0)
```

```
Before batch normalization:
  means:   [ -2.3814598  -13.18038246   1.91780462]
  stds:    [27.18502186 34.21455511 37.68611762]

After batch normalization (gamma=1, beta=0)

AttributeError: 'NoneType' object has no attribute 'mean'
```

然后发现原因是对于前向传播的函数编写不正确，从而导致了报错经过对前向传播的函数纠正，运行出了正确结果；