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：Regularization and Batch Normalization		学号：201900130151
日期：2021.10.23	班级： 人工智能	姓名： 莫甫龙
Email：m1533979510@163.com		

实验目的：

完成 Regularization

完成 Batch Normalization

实验软件和硬件环境：

Vs code

Win11

实验原理和方法：

Regularization:

L2 正则化：

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

Dropout：

假设我们要训练这样一个神经网络，如图2所示。

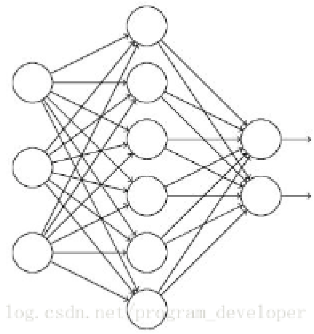

图2：标准的神经网络

输入是x输出是y，正常的流程是：我们首先把x通过网络前向传播，然后把误差反向传播以决定如何更新参数让网络进行学习。使用Dropout之后，过程变成如下：

（1）首先随机（临时）删掉网络中一半的隐藏神经元，输入输出神经元保持不变（图3中虚线为部分临时被删除的神经元）

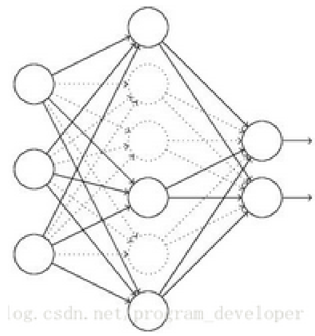

图3：部分临时被删除的神经元

（2） 然后把输入x通过修改后的网络前向传播，然后把得到的损失结果通过修改的网络反向传播。一小批训练样本执行完这个过程后，在没有被删除的神经元上按照随机梯度下降法更新对应的参数（w，b）。

（3）然后继续重复这一过程：

. 恢复被删掉的神经元（此时被删除的神经元保持原样，而没有被删除的神经元已经有所更新）

. 从隐藏层神经元中随机选择一个一半大小的子集临时删除掉（备份被删除神经元的参数）。

. 对一小批训练样本，先前向传播然后反向传播损失并根据随机梯度下降法更新参数（w，b） （没有被删除的那一部分参数得到更新，删除的神经元参数保持被删除前的结果）。

不断重复这一过程。

Batch Normalization:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

1 $\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ // mini-batch mean

2 $\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$ // mini-batch variance

3 $\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$ // normalize

4 $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)$ // scale and shift

$x_i \to \mu_B \to \sigma_B^2 \to \hat{x}_i \to y_i \to l$

$\therefore \frac{\partial l}{\partial \hat{x}_i} = \frac{\partial l}{\partial y_i}\cdot\frac{\partial y_i}{\partial \hat{x}_i} = \frac{\partial l}{\partial y_i}\cdot\gamma$

$\frac{\partial l}{\partial \beta} = \sum_i \frac{\partial l}{\partial y_i}\cdot\frac{\partial y_i}{\partial \beta} = \sum_i \frac{\partial l}{\partial y_i}$

$\frac{\partial l}{\partial \gamma} = \sum_i \frac{\partial l}{\partial y_i}\cdot\frac{\partial y_i}{\partial \gamma} = \sum_i \frac{\partial l}{\partial y_i}\cdot\hat{x}_i$

$\frac{\partial l}{\partial \sigma_B^2} = \sum_i \frac{\partial l}{\partial \hat{x}_i}\cdot\frac{\partial \hat{x}_i}{\partial \sigma_B^2} = \sum_i \frac{\partial l}{\partial \hat{x}_i}(-\frac{1}{2})(x_i-\mu_B)(\sigma_B^2+\epsilon)^{-\frac{3}{2}}$

$\frac{\partial l}{\partial \mu_B} = \sum_i(\frac{\partial l}{\partial \hat{x}_i}\cdot\frac{\partial \hat{x}_i}{\partial \mu_B} + \frac{\partial l}{\partial \sigma_B^2}\cdot\frac{\partial \sigma_B^2}{\partial \mu_B}) = \sum_i[\frac{\partial l}{\partial \hat{x}_i}\cdot(\frac{-1}{\sqrt{\sigma_B^2+\epsilon}})] + \frac{\partial l}{\partial \sigma_B^2}\cdot\frac{-2}{m}[\sum_i(x_i-\mu_B)]$

$= \sum_i \frac{\partial l}{\partial \hat{x}_i}\cdot(\frac{-1}{\sqrt{\sigma_B^2+\epsilon}})$

$\frac{\partial l}{\partial x_i} = \frac{\partial l}{\partial \hat{x}_i}\cdot\frac{\partial \hat{x}_i}{\partial x_i} + \frac{\partial l}{\partial \sigma_B^2}\cdot\frac{\partial \sigma_B^2}{\partial x_i} + \frac{\partial l}{\partial \mu_B}\cdot\frac{\partial \mu_B}{\partial x_i}$

$= \frac{\partial l}{\partial \hat{x}_i}\cdot\frac{1}{\sqrt{\sigma_B^2+\epsilon}} + \frac{\partial l}{\partial \sigma_B^2}\cdot\frac{2}{m}(x_i-\mu_B) + \frac{\partial l}{\partial \mu_B}\cdot\frac{1}{m}$

加速 $= \frac{\partial l}{\partial \hat{x}_i}\cdot(\sigma_B^2+\epsilon)^{-\frac{1}{2}} + \sum_j \frac{\partial l}{\partial \hat{x}_j}(-\frac{1}{2})(x_j-\mu_B)(\sigma_B^2+\epsilon)^{-\frac{3}{2}}\cdot\frac{2}{m}(x_i-\mu_B) + \sum_j \frac{\partial l}{\partial \hat{x}_j}\cdot(\frac{(\sigma_B^2+\epsilon)^{-\frac{1}{2}}}{-m})$

$= \frac{\partial l}{\partial \hat{x}_i}\cdot(\sigma_B^2+\epsilon)^{-\frac{1}{2}} - \sum_j \frac{\partial l}{\partial \hat{x}_j}\cdot\underbrace{\frac{x_j-\mu_B}{\sqrt{\sigma_B^2+\epsilon}}}_{\hat{x}_j}\cdot\underbrace{\frac{x_i-\mu_B}{\sqrt{\sigma_B^2+\epsilon}}}_{\hat{x}_i}\cdot\frac{(\sigma_B^2+\epsilon)^{-\frac{1}{2}}}{m} - \frac{(\sigma_B^2+\epsilon)^{-\frac{1}{2}}}{m}\cdot\sum_j \frac{\partial l}{\partial \hat{x}_j}$

$= \frac{(\sigma_B^2+\epsilon)^{-\frac{1}{2}}}{m}(m\cdot\frac{\partial l}{\partial \hat{x}_i} - \hat{x}_i\sum_j \frac{\partial l}{\partial \hat{x}_j}\cdot\hat{x}_j - \sum_j \frac{\partial l}{\partial \hat{x}_j})$

实验步骤：（不要求罗列完整源代码）

1. Regularization：

L2 Regularization：

```python
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
```

```python
### 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}

return gradients
```

Dropout：

```python
# 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 des
D1 = np.random.rand(A1.shape[0],A1.shape[1])                                        # Step 1: initializ
D1 = np.where(D1 <= keep_prob, 1, 0)                                                # Step 2: convert e
A1 = A1*D1                                          # Step 3: shut down some neurons of A1
A1 = A1/keep_prob                                          # Step 4: scale the value of neurons that
### 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: init
D2 = np.where(D2 <= keep_prob, 1, 0)                                            # Step 2: convert entr
A2 = A2*D2                                          # Step 3: shut down some neurons of A2
A2 = A2/keep_prob                                         # Step 4: scale the value of neurons that
### 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
```

```python
m = X.shape[1]
(Z1, D1, A1, W1, b1, Z2, D2, A2, W2, b2, Z3, A3, W3, b3) = 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. Batch Normalization:

batchnorm_forward:

```python
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    m = np.mean(x, axis=0)
    sigma2 = np.var(x, axis=0)
    x_ = (x - m) / np.sqrt(sigma2 + eps)
    out = gamma * x_ + beta
    sigma_2 = 1. / np.sqrt(sigma2 + eps)
    cache = (x, x_, gamma, m, sigma_2)
    running_mean = momentum * running_mean + (1 - momentum) * m
    running_var = momentum * running_var + (1 - momentum) * sigma2
    pass

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    #######################################################################
    #                           END OF YOUR CODE                          #
    #######################################################################
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_ = (x - running_mean) / np.sqrt(running_var + eps)
    out = gamma * x_ + beta
    pass

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

batchnorm_backward:

```python
    - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
    - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
    """
    dx, dgamma, dbeta = None, None, None
    ###########################################################################
    # 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_, gamma, m, sigma_2 = cache
    N = x.shape[0]
    dbeta = np.sum(dout, axis=0)
    dgamma = np.sum(x_ * dout, axis=0)
    dvar = np.sum(-0.5 * sigma_2 ** 3 * (x - m) * gamma * dout, axis=0)
    dm = np.sum(-1 * sigma_2 * gamma * dout, axis=0)
    dx = gamma * dout * sigma_2 + (2 / N) * (x - m) * dvar + (1 / N) * dm
    pass

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

    return dx, dgamma, dbeta
```

batchnorm_backward_alt:

```python
    Inputs / outputs: Same as batchnorm_backward
    """
    dx, dgamma, dbeta = None, None, None
    ###########################################################################
    # 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_, gamma,m, sigma_2 = cache
    N = x.shape[0]
    dbeta = np.sum(dout, axis=0)
    dgamma = np.sum(x_ * dout, axis=0)
    dxhat = dout * gamma
    dx = (1. / N) * sigma_2 * (N * dxhat - np.sum(dxhat, axis=0) - x_ * np.sum(dxhat * x_, axis=0))
    pass

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

加速结果：

```
    print('speedup: %.2fx' % ((t2 - t1) / (t3 - t2)))
[28]
...  dx difference:  7.494857050222097e-13
     dgamma difference:  0.0
     dbeta difference:  0.0
     speedup: 2.00x
```

结论分析与体会：

该实验主要分为正则项和 BN 两部分，正则项还好，只需要前向和反向传播就行，但是 BN 的反向传播计算导数却比较复杂，并且在写加速的时候发现居然只要把原来的式子化简就行。

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：

在写 BN 的加速算法的时候，很懵，不知道从何下手，后面通过查阅资料发现它的加速就是在原来得到的式子的基础上给它化简就行，将多余的计算直接化简为一条简单的式子，从而达到加速。后面发现在运行的时候，它的这个加速速率是不定的，会随着数据集的变化而变化。