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：Improving Deep Neural Networks		学号：201900130151
日期：2021.10.10	班级：人工智能	姓名： 莫甫龙
Email：m1533979510@163.com		

实验目的：

熟悉和掌握神经网络中一些基础的初始化和优化

实验软件和硬件环境：

Vs code

Win11

实验原理和方法：

1. Initialization：

好的初始化加速梯度下降法的收敛，也可以增加梯度下降收敛到较低训练误差的几率，其中代码中给了三种初始化的方法：

Zeros initialization:将 W 和 b 全部初始化为 0，这会导致代价根本就没有变化。

Random initialization:将 W 初始化为较大的随机值，这种做法成本很高，并且还会导致梯度消失或者梯度爆炸，并且还会减慢优化的速度。

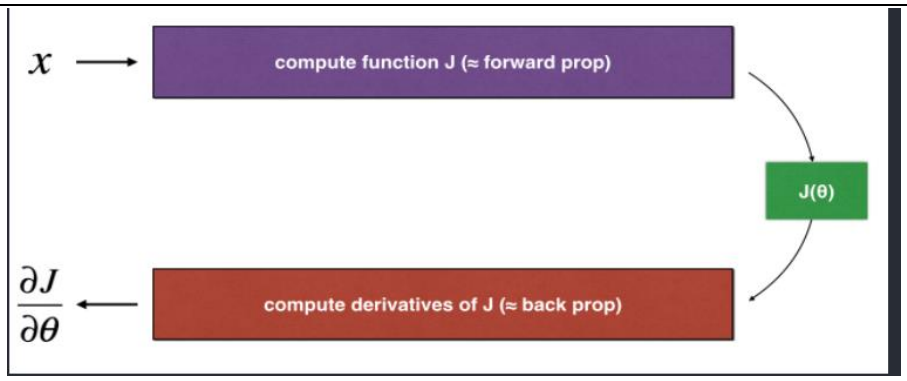
He initialization:将 W 初始化为较小的随机值，可以在较少的迭代中表现出很好的效果。

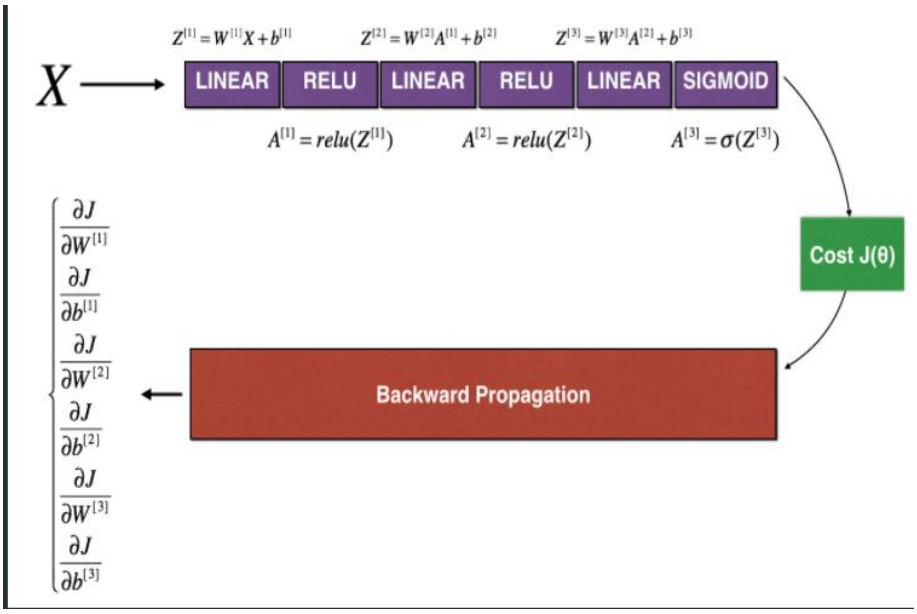
2. Gradient Checking：

计算梯度：

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

正向传播和反向传播：





3. Optimization:

1- Gradient Descent:

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

2- Mini-batch Gradient Descent:

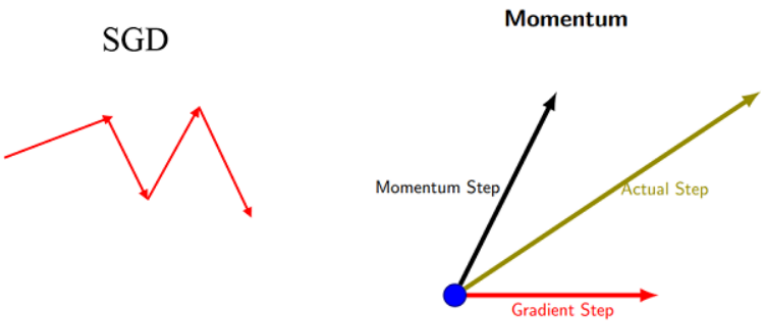
在上述的批梯度的方式中每次迭代都要使用到所有的样本，对于数据量特别大的情况，如大规模的机器学习应用，每次迭代求解样本需要花费大量的计算成本。是否可以在每次的迭代过程中利用部分样本代替所有的样本呢？基于这样的思想，便出现了mini-batch概念。

假设训练集中的样本的个数为1000，则每个mini-batch只是其一个子集，假设，每个mini-batch中含有10个样本，这样，整个训据集可以分为100个mini-batch。伪代码如下：

```
Repeat{
    for i=1, 11, 21, 31, ... , 991{
    θ_j := θ_j − α (1/10) Σ_{k=i}^{i+9} (h_θ(x^(k)) − y^(k)) x_j^(k)
    (for every j=0, ... , n)
    }
}
```

使用 mini-batch 来进行遍历训练集，在一次遍历中，可以得到很多次的梯度下降，因为它可以看成是并行运行的。

3- Momentum：



区别： SGD每次都会在当前位置上沿着负梯度方向更新（下降，沿着正梯度则为上升），并不考虑之前的方向梯度大小等等。而（moment）通过引入一个新的变量 v 去积累之前的梯度（通过指数衰减平均得到），得到加速学习过程的目的。

> 最直观的理解就是，若当前的梯度方向与累积的历史梯度方向一致，则当前的梯度会被加强，从而这一步下降的幅度更大。若的梯度方向与累积的梯度方向不一致，则会减弱当前下降的梯度幅度。

4- Adam：

Adam与经典的随机梯度下降法是不同的。随机梯度下降保持一个单一的学习速率(称为alpha)，用于所有的权重更新，并且在训练过
学习速率不会改变。每一个网络权重(参数)都保持一个学习速率，并随着学习的展开而单独地进行调整。该方法从梯度的第一次和第
矩的预算来计算不同参数的自适应学习速率。

作者描述Adam时将随机梯度下降法两种扩展的优势结合在一起。

具体地说:

自适应梯度算法(AdaGrad)维护一个参数的学习速率，可以提高在稀疏梯度问题上的性能(例如，自然语言和计算机视觉问题)。

均方根传播(RMSProp)也维护每个参数的学习速率，根据最近的权重梯度的平均值(例如变化的速度)来调整。这意味着该算法在线上
平稳问题上表现良好(如:噪声)。

Adam意识到AdaGrad和RMSProp的好处。与在RMSProp中基于平均第一个时刻(平均值)的参数学习速率不同，Adam也使用了梯度
二个时刻的平均值(非中心方差)。

具体地说，该算法计算了梯度和平方梯度的指数移动平均值，并且参数beta1和beta2控制了这些移动平均的衰减率。移动平均值和beta1
和beta2的初始值接近1.0(推荐值)，这导致了估计时间的偏差为0。这种偏差是通过第一次计算偏差估计然后再计算比可用偏差校正
克服的。

实验步骤：（不要求罗列完整源代码）

1. Initialization：

Zeros initialization：

```
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
```

Random initialization:

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

return parameters
```

He initialization:

```
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 ###

return parameters
```

2. Gradient Checking：

```
    # Compute gradapprox using left side of formula (1). epsilon is small enough, you don't need to worry about the li
    ### START CODE HERE ### (approx. 5 lines)
    thetaplus = theta + epsilon                    # Step 1
    thetaminus = theta - epsilon                   # Step 2
    J_plus =  thetaplus * x                    # Step 3
    J_minus =  thetaminus * x                    # 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 ###

    if difference < 1e-7:
        print ("The gradient is correct!")
    else:
        print ("The gradient is wrong!")

    return difference
```

```
# Compute gradapprox
for i in range(num_parameters):

    # 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 one
    ### 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))                                 # S
    ### 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))                               # S
    ### 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 ###
```

3.Optimization:

1.Gradient Descent:

```
    # Update rule for each parameter
    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 ###

    return parameters
✓ 0.2s
```

2. Mini-batch Gradient Descent：

```
# 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_batch_size in
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)

# 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[:, num_complete_minibatches*mini_batch_size:]
    mini_batch_Y = shuffled_Y[:, num_complete_minibatches*mini_batch_size:]
    ### END CODE HERE ###
    mini_batch = (mini_batch_X, mini_batch_Y)
    mini_batches.append(mini_batch)

return mini_batches
```

3. Momentum：

```
    # 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 ###

    return v
```

```
  # 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)] = parameters["W" + str(l+1)] - learning_rate * v["dW" + str(l+1)]
      parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - learning_rate * v["db" + str(l+1)]
      ### END CODE HERE ###

  return parameters, v
```

4. Adam：

```
    # Initialize v, s. Input: "parameters". Outputs: "v, s".
    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 ###

    return v, s
```

```
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)*np.power(grads['dW' + str(l+1)], 2)
    s["db" + str(l+1)] = beta2 * s["db" + str(l+1)] + (1-beta2)*np.power(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)] = parameters["W" + str(l+1)] - learning_rate * v_corrected["dW" + str(l+1)] / (np.sqrt(s_corrected["
    parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - learning_rate * v_corrected["db" + str(l+1)] / (np.sqrt(s_corrected["
    ### END CODE HERE ###
```

结论分析与体会：

对于训练神经网络来说，初始化真的十分重要，初始化得过大或者过小都不行，对于不同的激活函数，合适的初始值也是不一样的，代码中的 He 是对 Relu 激活函数比较有效的。

而梯度下降的优化，代码中给出了 4 种不同的方法，各有各自的优点，有些虽然已经不是很适用于现在了，但是依旧十分具有学习和借鉴的意义。

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：

对于 Adam 算法的理解：

在随机梯度下降保持单一的学习率更新所有的权重，学习率在训练过程中并不会改变。而 Adam 通过随机梯度的一阶矩估计和二阶矩估计而为不同的参数设计独立的自适应性学习率。