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：201800130086
日期：2020-10-20	班级： 18 智能班	姓名：徐鹏博
Email：hsupengbo@163.com		

实验目的：

掌握神经网络中常用的一些调试、优化方法

• Hyperparameter tuning

• Regularization

• Optimization.

实验软件和硬件环境：

Anaconda3,

JupyterNotebook

Inter(R) Core(TM) i5-8250 CPU @1.60GHz 1.80GHz

Win10-x64

实验原理和方法：

•Q1: Initialization

好的初始化方法会加速梯度下降的收敛，增加 梯度下降收敛成一个低错误训练 的几率。

Zeros initialization 把 w 初始化为 0 是不可以的，代价没有真正的下降

Random initialization 随机初始化时，用过大的随机数初始化工作的不是很好，出现梯度消失/梯度爆炸，会减缓优化，找一个比较小的数随机初始化。

He initialization 可以解决梯度爆炸/梯度消失

适用于 ReLU 的初始化方法:

$$W \sim N[0, \sqrt{\frac{2}{n_i}}]$$

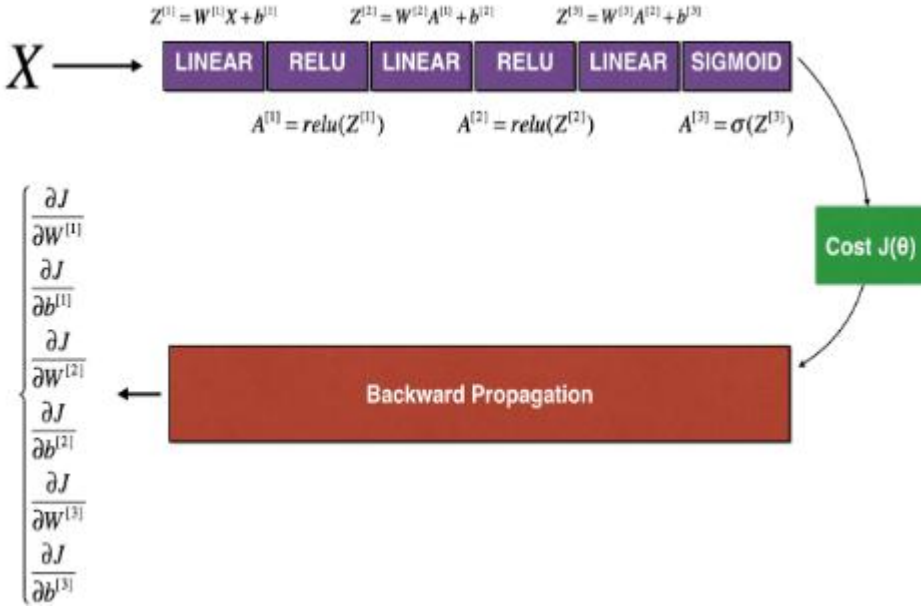
•Q2: Gradient Checking

Backpropagation 计算梯度(the gradients) :

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

1-dimensional gradient checking: J(θ)=θx, dθ=x

N-dimensional gradient checking:



•Q3: Optimization

Mini-batch 梯度下降

可以把训练集分割为小一点的子集进行训练，这些子集即为 mini-batch，假设每个子集中有 1000 个样本，那么把其中的 x(1)-x(1000)到取出来，称为第一个子训练集 X(1)，也叫做 mini-batch，然后以此类推取出其他部分，得到 X{5000}。对 Y 也做同样的拆分处理,得到 Y{5000}。mini-batch 的数量 t 组成了 X{t}和 Y{t}，这就是 1000 个训练样本，包含相应的输入输出对。首先对输入 X{t}执行前向传播,然后执行 Z[1]=W[1]X+b[1], A[1]=g[1](Z[1]), 以此类推到 A[L]=g[L](Z[L]),之后进行反向传播来计算梯度。使用 batch 梯度下降法，一次遍历训练集只能让你做一个梯度下降，使用 mini-batch 梯度下降法，一次遍历训练集，能让你做 5000 个梯度下降。一直处理遍历训练集，直到最后你能收敛到一个合适的精度。

Momentum

或者叫做动量梯度下降法，运行速度几乎总是快于标准的梯度下降算法，简而言之，基本的想法就是计算梯度的指数加权平均数，并利用该梯度更新你的权重。

Adam

Adam 优化算法基本上就是将 Momentum 和 RMSprop 结合在一起。Adam 使用动量和自适应学习率来加快收敛速度.动量更新方法，其中θ是网络的参数，即权重，偏差或激活值，η是学习率，J 是我们要优化的目标函数，γ是常数项，也称为动量。 Vt-1（注意 t-1 是下标）是过去的时间步长，而 Vt（注意 t 是下标）是当前的时间步长。

实验步骤：（不要求罗列完整源代码）

Zeros initialization

```
# GRADED FUNCTION: initialize_parameters_zeros
def initialize_parameters_zeros(layers_dims):
    parameters = {}
    L = len(layers_dims)            # number of layers in the network
    for l in range(1, L):
        parameters['W' + str(l)] = np.zeros((layers_dims[l], layers_dims[l-1]))
        parameters['b' + str(l)] = np.zeros((layers_dims[l], 1))
    return parameters
```

Random initialization

```
# GRADED FUNCTION: initialize_parameters_random
def initialize_parameters_random(layers_dims):
    np.random.seed(3)               # This seed makes sure your "random" numbers will be the as ours
    parameters = {}
    L = len(layers_dims)            # integer representing the number of layers

    for l in range(1, L):
        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))
    return parameters
```

He initialization

```
# GRADED FUNCTION: initialize_parameters_he
def initialize_parameters_he(layers_dims):
    np.random.seed(3)
    parameters = {}
    L = len(layers_dims) - 1 # integer representing the number of layers

    for l in range(1, 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((layers_dims[l], 1))
```

```
    return parameters
```

1-dimensional gradient checking

forward_propagation:

```
# GRADED FUNCTION: forward_propagation
def forward_propagation(x, theta):
    J = theta * x
    return J

# GRADED FUNCTION: backward_propagation
```

backward_propagation:

```
def backward_propagation(x, theta):
    dtheta = x
    return dtheta
```

gradient_check

```
# GRADED FUNCTION: gradient_check
def gradient_check(x, theta, epsilon = 1e-7):
    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
    grad = backward_propagation(x, theta)
    numerator =   np.linalg.norm(grad - gradapprox)                                 # Step 1'
    denominator =    np.linalg.norm(grad) + np.linalg.norm(gradapprox)                      # Step 2'
    difference = numerator / denominator                                    # Step 3'

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

N-dimensional gradient checking:

```
# GRADED FUNCTION: gradient_check_n
def gradient_check_n(parameters, gradients, X, Y, epsilon = 1e-7):
    # Set-up variables
    parameters_values, _ = dictionary_to_vector(parameters)
    grad = gradients_to_vector(gradients)
    num_parameters = parameters_values.shape[0]
    J_plus = np.zeros((num_parameters, 1))
    J_minus = np.zeros((num_parameters, 1))
    gradapprox = np.zeros((num_parameters, 1))
```

```
    # Compute gradapprox
    for i in range(num_parameters):
        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
        thetaminus = np.copy(parameters_values)                                     # Step 1
        thetaminus[i][0] = thetaplus[i][0] - epsilon                                # Step 2
        J_minus[i], _ = forward_propagation_n(X, Y, vector_to_dictionary(thetaminus))  # Step 3
        gradapprox[i] = (J_plus[i] - J_minus[i])/(2 * epsilon)
    numerator = np.linalg.norm(grad - gradapprox)                                   # Step 1'
    denominator = np.linalg.norm(grad) + np.linalg.norm(gradapprox)                 # Step 2'
    difference = numerator / denominator                                            # Step 3'

    if difference > 1e-7:
        print ("\033[93m" + "There is a mistake in the backward propagation! difference = " + str(difference) +
"\033[0m")
    else:
        print ("\033[92m" + "Your backward propagation works perfectly fine! difference = " + str(difference) +
"\033[0m")
    return difference
```

Mini-batch 梯度下降

```
# GRADED FUNCTION: random_mini_batches
def random_mini_batches(X, Y, mini_batch_size = 64, seed = 0):
    np.random.seed(seed)            # To make your "random" minibatches the same as ours
    m = X.shape[1]                  # number of training examples
    mini_batches = []

    # Step 1: Shuffle (X, Y)
    permutation = list(np.random.permutation(m))
    shuffled_X = X[:, permutation]
    shuffled_Y = Y[:, permutation].reshape((1,m))

    num_complete_minibatches = math.floor(m/mini_batch_size)
    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
```

Momentum

```
# GRADED FUNCTION: initialize_velocity
def initialize_velocity(parameters):
    L = len(parameters) // 2 # number of layers in the neural networks
    v = {}

    for l in range(L):
        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)])
    return v

# GRADED FUNCTION: update_parameters_with_momentum
def update_parameters_with_momentum(parameters, grads, v, beta, learning_rate):
    L = len(parameters) // 2 # number of layers in the neural networks

    # Momentum update for each parameter
    for l in range(L):
        # 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)]

    return parameters, v
```

Adam

```
# GRADED FUNCTION: initialize_adam
def initialize_adam(parameters) :
    L = len(parameters) // 2 # number of layers in the neural networks
    v = {}
    s = {}

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

```
    for l in range(L):
        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)])

    return v, s

# GRADED FUNCTION: update_parameters_with_adam
def update_parameters_with_adam(parameters, grads, v, s, t, learning_rate = 0.01,
                                beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8):
    L = len(parameters) // 2                 # number of layers in the neural networks
    v_corrected = {}                         # Initializing first moment estimate, python dictionary
    s_corrected = {}                         # Initializing second moment estimate, python dictionary

    # Perform Adam update on all parameters
    for l in range(L):
        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)]

        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))

        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)

        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))
        parameters["W" + str(l+1)] = 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)] = parameters["b" + str(l+1)] - learning_rate
                                     * ( v_corrected["db" + str(l+1)]
                                     / (np.sqrt(s_corrected["db" + str(l+1)]) + epsilon))
    return parameters, v, s
```

结论分析与体会：

不同初始化会有不同的结果。
随机初始化能打破对称，但是不能用太大的数来初始化。

He initialization 比较适合 ReLU 的初始化。

梯度检测比较耗时,所以在训练集上不能每次迭代都检测，需要及时关掉。
使用 batch 梯度下降法每次迭代都需要历遍整个训练集。可以预期每次迭代成本都会下降，
所以如果成本函数是迭代次数的一个函数，它应该会随着每次迭代而减少，如果在某次迭代中增加可能是因为学习率太大。

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：

如何理解动量梯度下降法:

梯度下降法每次的参数更新公式：

$$W := W - \alpha \nabla W$$

$$b := b - \alpha \nabla b$$

每次更新仅与当前梯度值相关，并不涉及之前的梯度。而动量梯度下降法则对各个 mini-batch 求得的梯度▿W,▿b 使用指数加权平均得到 V▿W,v▿b 并使用新的参数更新之前的参数。