改善深层神经网络-算法总结

赵燕

目录

1 Practical aspects of Deep Learning

1.1 Initialization

一个好的初始化可以加快梯度下降的收敛，同时能够以较大几率使得梯度下降收敛到较低的训练和泛化误差。

使用一个三层的神经网络去实现初始化，共有三种方法：

1.全零初始化（Zero Initialization）

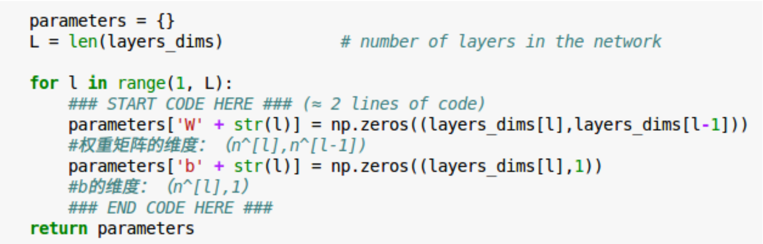
2.随机初始化（Random Initialization）

3.He初始化（He Initialization）

Initialize parameters dictionary:

```
if initialization == "zeros":
    parameters=initialize_parameters_zeros(layers_dims)
elif initialization == "random":
    parameters=initialize_parameters_random(layers_dims)
elif initialization == "he":
    parameters=initialize_parameters_he(layers_dims)
```

1.1.1 Zero Initialization



```
parameters = {}
L = len(layers_dims)            # number of layers in the network

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]))
    #权重矩阵的维度: (n^[l],n^[l-1])
    parameters['b' + str(l)] = np.zeros((layers_dims[l],1))
    #b的维度: (n^[l],1)
    ### END CODE HERE ###
return parameters
```

图 1: 全零初始化

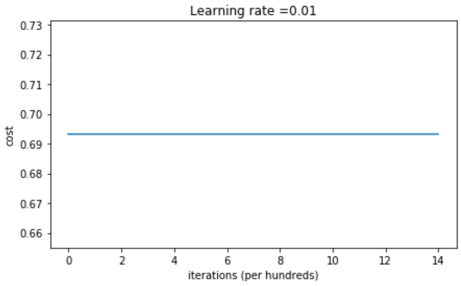


图 2: 全零初始化代价函数曲线

全零初始化的效果很差，整个迭代过程中损失函数并不会下降，保持一条直线的状态，这个参数下的模型，不管是对于训练集还是测试集，预测结果都为0。

通常，将所有权重初始化为零会导致网络不能破坏对称性。这意味着每一层的每个神经元都会学习同样的东西，你也可以训练一个神经网络，每个层都有$n^{[l]} = 1$，那神经网络的性能就如线性分类器，比如说逻辑回归。

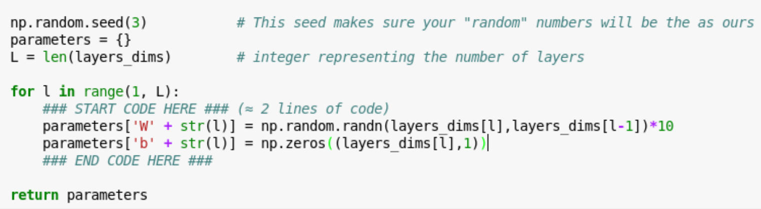
对于参数$W^{[l]}$，我们用进行随机初始化，以打破神经网络的对称性。对于参数$b^{[l]}$是可以初始化为0的。

1.1.2 Random Initialization

为了打破对称，使用随机初始化权重。随机初始化后，每个神经元然后可以继续学习其输入的不同功能。

Use np.random.randn(..,..) * 10 for weights and np.zeros((.., ..)) for biases.

We are using a fixed np.random.seed(..) to make sure your ”random” weights match ours, so don’t worry if running several times your code gives you always the same initial values for the parameters.



```
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):
    ### 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
```

图 3: 随机初始化

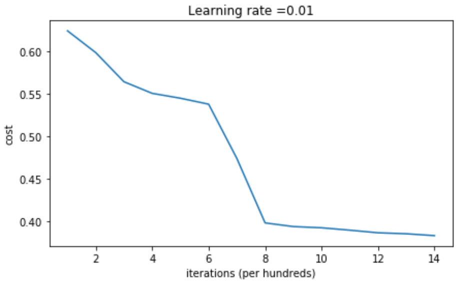


图 4: 随机初始化代价函数曲线

一开始的代价函数很大,这是由于采用的随机初始化的权重矩阵值较大,导致有些样本对于激活函数(sigmoid,输出层)输出结果值很靠近 0 或者 1。当输出的结果值不同于真实值,其代价很大,比如log($a^{[3]}$)=log(0),代价是无穷大的。

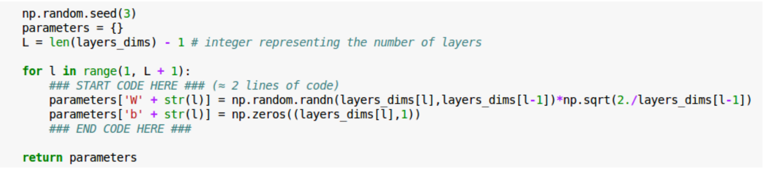
过大或者过小的权重矩阵将导致梯度爆炸或梯度消失,这都会减缓优化算法获取最优结果的速度。因此需要控制随机化的权重矩阵大小。

如果更长时间的训练网络，将会看到更好的结果，但是用过大的随机数进行初始化会降低优化速度。

总结：将权重初始化为非常大的随机值无效，所以要用小的随机值进行初始化。

1.1.3 He Initialization

He初始化方式是对随机初始化的权重矩阵乘以$sqrt(2./layers_dims[l-1])$;Xavier方式初始化是对权重矩阵乘以$sqrt(1./layers_dims[l-1])$。



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

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
```

图 5: He初始化

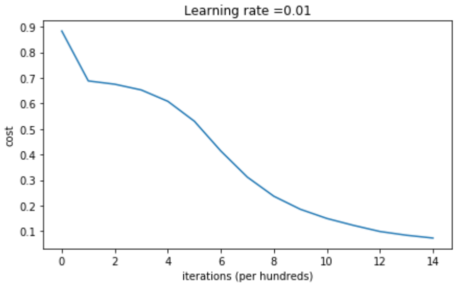
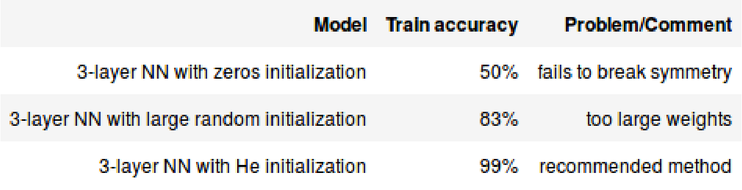


图 6: He初始化代价函数曲线

You have seen three different types of initializations. For the same number of iterations and same hyperparameters the comparison is:



Model	Train accuracy	Problem/Comment
3-layer NN with zeros initialization	50%	fails to break symmetry
3-layer NN with large random initialization	83%	too large weights
3-layer NN with He initialization	99%	recommended method

图 7: 初始化方式总结

1.2 Regularization

不使用正则化会有明显的过拟合现象，为了避免过度拟合现象，引入了两种正则化技术：

- L_2 regularization

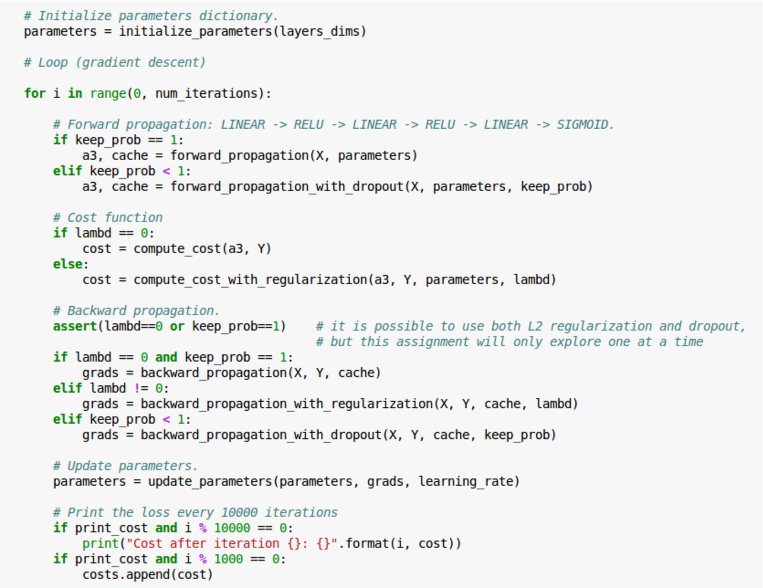
  参数：λ，在Python中lambda是保留字，所以此时的 λ 写作：”lambd”

  函数：”compute_cost_with_regularization()” 和 ”backward_propagation_with_regularization()”

- Dropout regularization

  参数:设置keep_prob值，作为神经元结点的概率。

函数："forward_propagation_with_dropout()" and "backward_propagation_with_dropout()"



```
# Initialize parameters dictionary.
parameters = initialize_parameters(layers_dims)

# Loop (gradient descent)

for i in range(0, num_iterations):

    # Forward propagation: LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID.
    if keep_prob == 1:
        a3, cache = forward_propagation(X, parameters)
    elif keep_prob < 1:
        a3, cache = forward_propagation_with_dropout(X, parameters, keep_prob)

    # Cost function
    if lambd == 0:
        cost = compute_cost(a3, Y)
    else:
        cost = compute_cost_with_regularization(a3, Y, parameters, lambd)

    # Backward propagation.
    assert(lambd==0 or keep_prob==1)    # it is possible to use both L2 regularization and dropout,
                                        # but this assignment will only explore one at a time
    if lambd == 0 and keep_prob == 1:
        grads = backward_propagation(X, Y, cache)
    elif lambd != 0:
        grads = backward_propagation_with_regularization(X, Y, cache, lambd)
    elif keep_prob < 1:
        grads = backward_propagation_with_dropout(X, Y, cache, keep_prob)

    # Update parameters.
    parameters = update_parameters(parameters, grads, learning_rate)

    # Print the loss every 10000 iterations
    if print_cost and i % 10000 == 0:
        print("Cost after iteration {}: {}".format(i, cost))
    if print_cost and i % 1000 == 0:
        costs.append(cost)
```

图 8: 正则化函数

1.2.1 L_2 Regularization

公式：

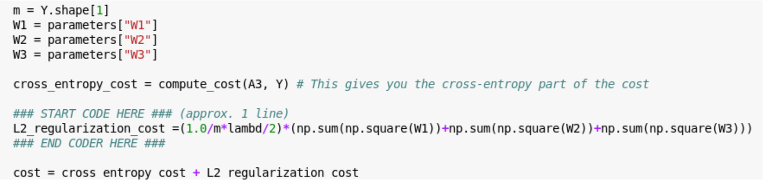
$$J = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}log(a^{[L](i)}) + (1-y^{(i)})log(1-a^{[L](i)})) \tag{1}$$

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}log(a^{[L](i)}) + (1-y^{(i)})log(1-a^{[L](i)}))}_{\text{corss-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l](2)}}_{L_2\text{ regularization cost}} \tag{2}$$

首先定义函数compute_cost_with_regularization()，表示公式（2）的代价函数。

$\sum\limits_k\sum\limits_j W_{k,j}^{[l]2}$的计算,采用:np.sum(np.square(Wl))

Note that you have to do this for $W^{[1]}$, $W^{[2]}$ and $W^{[3]}$, then sum the three terms and multiply by $\frac{1}{m}\frac{1}{2}$.



```
m = Y.shape[1]
W1 = parameters["W1"]
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 =(1.0/m*lambd/2)*(np.sum(np.square(W1))+np.sum(np.square(W2))+np.sum(np.square(W3)))
### END CODER HERE ###

cost = cross_entropy_cost + L2_regularization_cost
```

图 9: L_2正则化

由于代价函数改变了，所以同样需要修改后向传播算法，，则需要计算dW1, dW2 and dW3. 添加梯度下降的正则化项：$(\frac{d}{dW}(\frac{1}{2}\frac{\lambda}{m}W^2) = \frac{\lambda}{m}W)$.

设置λ的值等于0.7时：

```
parameters = model(train_X,train_Y, lambd = 0.7)
```

- The value of λ is a hyperparameter that you can tune using a dev set.
- L2 regularization makes your decision boundary smoother. If λ is too large, it is also possible to "oversmooth", resulting in a model with high bias.

L2-regularization relies on the assumption that a model with small weights is simpler than a model with large weights. Thus, by penalizing the square values of the weights in the cost function you drive all the weights to smaller values. It becomes too costly for the cost to have large weights! This leads to a smoother model in which the output changes more slowly as the input changes.

1.2.2 Dropout Regularization

Dropout正则化会遍历网络的每一层，并设置消除神经网络中节点的概率，消除一些节点，删除从该节点进出的连线，节点更少，则变成了规模更小的网络，然后用backprop方法进行训练，注意Dropout正则化仅仅适用于训练集！

前向传播算法:

You would like to shut down some neurons in the first and second layers. To do that, you are going to carry out 4 Steps:

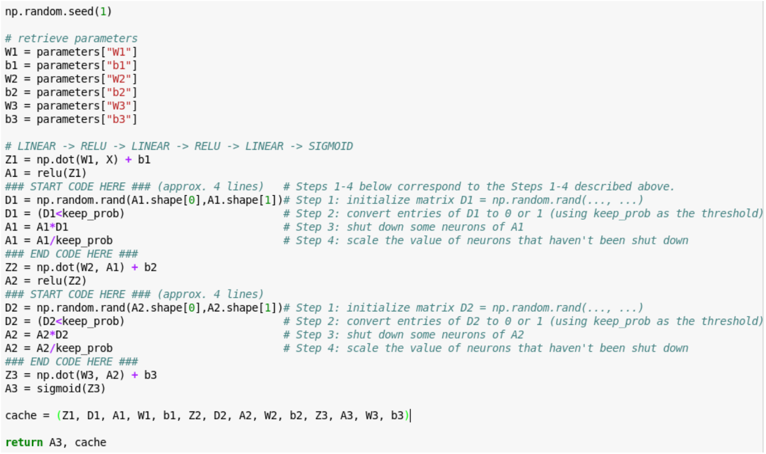
1. $d^{[1]}$ 与 $a^{[1]}$ 维度相同，使用 'np.random.rand()'进行初始化，同样随机初始化矩阵 $D^{[1]} = [d^{[1](1)} d^{[1](2)} ... d^{[1](m)}]$，和初始化 $A^{[1]}$的式一样。

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.$A^{[1]}$=$A^{[1]} * D^{[1]}$.

4. $A^{[1]}$/keep_prob.(inverted dropout.)



```
np.random.seed(1)

# retrieve parameters
W1 = parameters["W1"]
b1 = parameters["b1"]
W2 = parameters["W2"]
b2 = parameters["b2"]
W3 = parameters["W3"]
b3 = parameters["b3"]

# 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 = (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)

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

return A3, cache
```

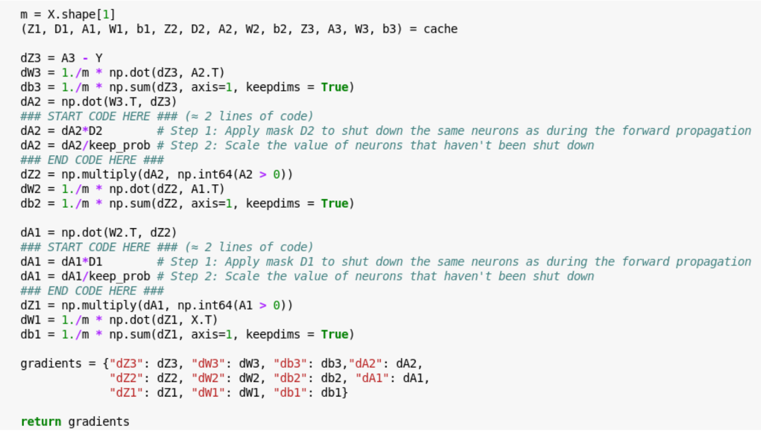
图 10: Dropout正则化前向传播算法

后向传播算法:

分两步:

1.前向传播过程中 ‘A1‘用到的$D^{[1]}$,在后向传播过程将$D^{[1]}$ 应用到dA1即可。

2.前向传播过程将 A1 除以 keep_prob ,以保持期望值。在后向传播过程 dA1 也做如此操作,即dA1/keep_prob 。这是由于$A^{[1]}$被 keep_prob 进行一定程度的放大,则其导数$dA^{[1]}$ 也需要做同比例的操作。

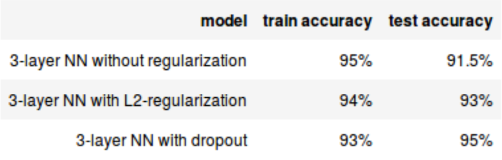


```
m = X.shape[1]
(Z1, D1, A1, W1, b1, Z2, D2, A2, W2, b2, Z3, A3, W3, b3) = cache

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
```

图 11: Dropout正则化前向传播算法

注意：Dropout正则化仅仅用于训练集！！！

总结:



model	train accuracy	test accuracy
3-layer NN without regularization	95%	91.5%
3-layer NN with L2-regularization	94%	93%
3-layer NN with dropout	93%	95%

图 12: 正则化方式对比

1.3 Gradient Checking

在神经网络计算过程中,对后向传播的梯度要进行校验,确保其计算无误。前向传播一般相对简单不会出错,所以在前向传播的基础上利用计算出来的代价J我们可以进行后向梯度的校验。

定义:

$$\frac{\partial J}{\partial \theta} = \lim_{\varepsilon \to 0} \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2\varepsilon} \tag{1}$$

- 计算$\frac{\partial J}{\partial \theta}$
- 计算$J(\theta + \varepsilon)$ and $J(\theta - \varepsilon)$

1.3.1 1-dimensional gradient checking

（1）计算梯度近似值:

1. $\theta^{+} = \theta + \varepsilon$

2. $\theta^{-} = \theta - \varepsilon$

3. $J^+ = J(\theta^+)$

4. $J^- = J(\theta^-)$

5. $gradapprox = \frac{J^+ - J^-}{2\varepsilon}$

（2）计算梯度grad

（3）计算difference:

$$difference = \frac{\| grad - gradapprox \|_2}{\| grad \|_2 + \| gradapprox \|_2} \tag{2}$$

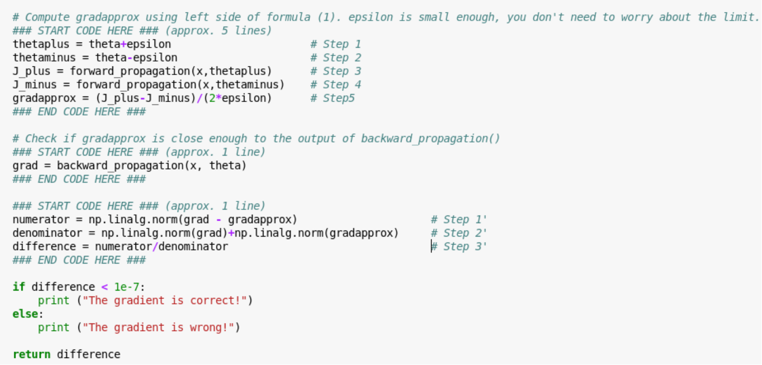
计算公式:

1.计算分子：np.linalg.norm(...)(范数)

2.计算分母：np.linalg.norm(...) twice.

3.相除

当difference足够小（$< 10^{-7}$），则视为通过梯度校验。



```python
# Compute gradapprox using left side of formula (1). epsilon is small enough, you don't need to worry about the limit.
### 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)         # Step5
### 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
```

图 13: 梯度检验

1.3.2 N-dimensional gradient checking

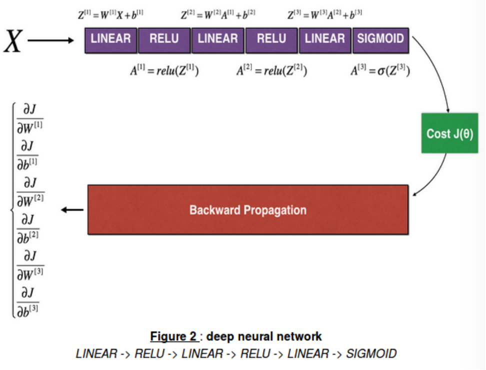
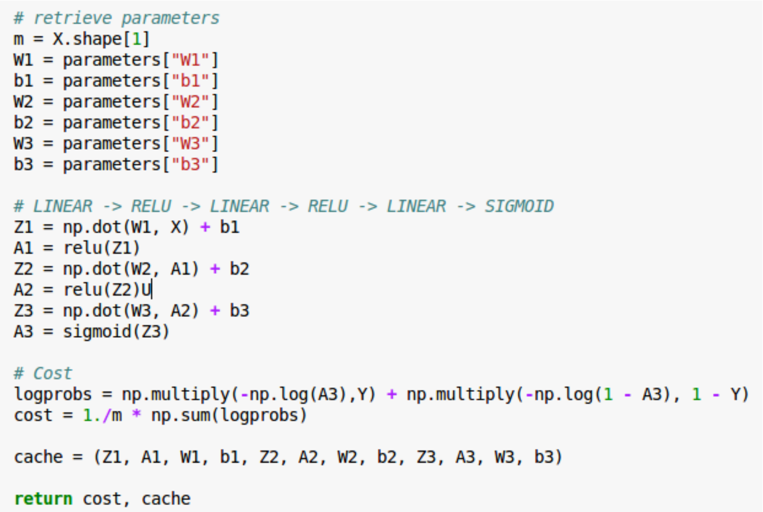

$Z^{[1]} = W^{[1]}X + b^{[1]}$
$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$
$Z^{[3]} = W^{[3]}A^{[2]} + b^{[3]}$
X
LINEAR
RELU
LINEAR
RELU
LINEAR
SIGMOID
$A^{[1]} = relu(Z^{[1]})$
$A^{[2]} = relu(Z^{[2]})$
$A^{[3]} = \sigma(Z^{[3]})$
Cost J(θ)
Backward Propagation
$\frac{\partial J}{\partial W^{[1]}}$ $\frac{\partial J}{\partial b^{[1]}}$ $\frac{\partial J}{\partial W^{[2]}}$ $\frac{\partial J}{\partial b^{[2]}}$ $\frac{\partial J}{\partial W^{[3]}}$ $\frac{\partial J}{\partial b^{[3]}}$
Figure 2 : deep neural network
LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID

图 14: 三层神经网络前向和后向

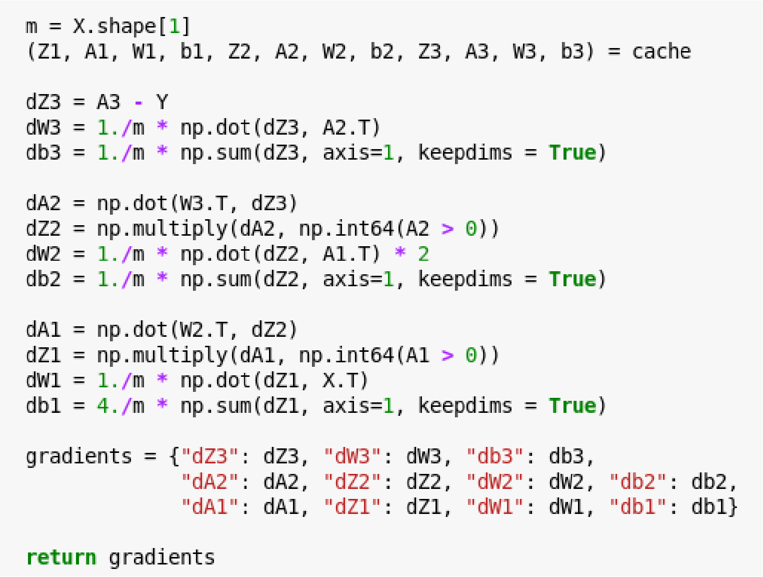
前向传播算法:



```
# retrieve parameters
m = X.shape[1]
W1 = parameters["W1"]
b1 = parameters["b1"]
W2 = parameters["W2"]
b2 = parameters["b2"]
W3 = parameters["W3"]
b3 = parameters["b3"]

# LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID
Z1 = np.dot(W1, X) + b1
A1 = relu(Z1)
Z2 = np.dot(W2, A1) + b2
A2 = relu(Z2)U
Z3 = np.dot(W3, A2) + b3
A3 = sigmoid(Z3)

# Cost
logprobs = np.multiply(-np.log(A3),Y) + np.multiply(-np.log(1 - A3), 1 - Y)
cost = 1./m * np.sum(logprobs)

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

return cost, cache
```

图 15: 前向传播算法

后向传播算法:



```
m = X.shape[1]
(Z1, A1, W1, b1, Z2, 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)
dZ2 = np.multiply(dA2, np.int64(A2 > 0))
dW2 = 1./m * np.dot(dZ2, A1.T) * 2
db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)

dA1 = np.dot(W2.T, dZ2)
dZ1 = np.multiply(dA1, np.int64(A1 > 0))
dW1 = 1./m * np.dot(dZ1, X.T)
db1 = 4./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
```

图 16: 后向传播算法

N维梯度检验:

For each i in num_parameters:

- To compute ‘J_plus[i]‘:

  1. Set θ^{+} to ‘np.copy(parameters_values)‘

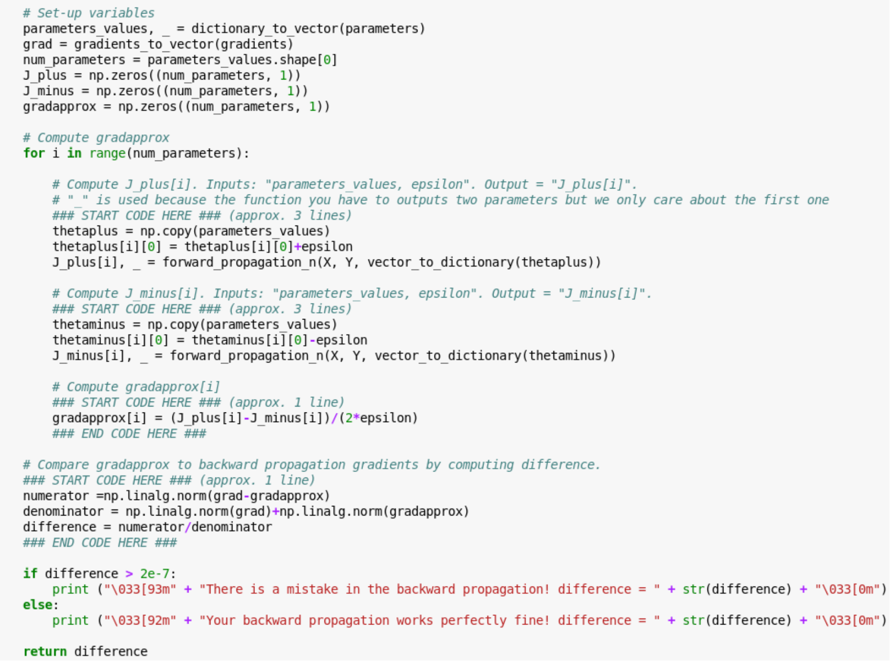
  2. Set θ_i^{+} to $\theta_i^{+} + \varepsilon$

  3. Calculate J_i^{+} using to ‘forward_propagation_n(x, y, vector_to_dictionary(‘θ^{+} ‘))‘.

- To compute ‘J_minus[i]‘: do the same thing with θ^{-}

- Compute $gradapprox[i] = \frac{J_i^+ - J_i^-}{2\varepsilon}$

  然后再计算difference



```
# 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):

    # 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)
    thetaplus[i][0] = thetaplus[i][0]+epsilon
    J_plus[i], _ = forward_propagation_n(X, Y, vector_to_dictionary(thetaplus))

    # Compute J_minus[i]. Inputs: "parameters_values, epsilon". Output = "J_minus[i]".
    ### START CODE HERE ### (approx. 3 lines)
    thetaminus = np.copy(parameters_values)
    thetaminus[i][0] = thetaminus[i][0]-epsilon
    J_minus[i], _ = forward_propagation_n(X, Y, vector_to_dictionary(thetaminus))

    # 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)
denominator = np.linalg.norm(grad)+np.linalg.norm(gradapprox)
difference = numerator/denominator
### END CODE HERE ###

if difference > 2e-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
```

图 17: 梯度检验

注意：如果修改代码，则需要重新执行定义backward_propagation_n（）的单元格。

梯度检验很慢，所以我们不会在训练期间每一次都运行梯度检验，只是几次检查梯度即可。

2 Optimization algorithms

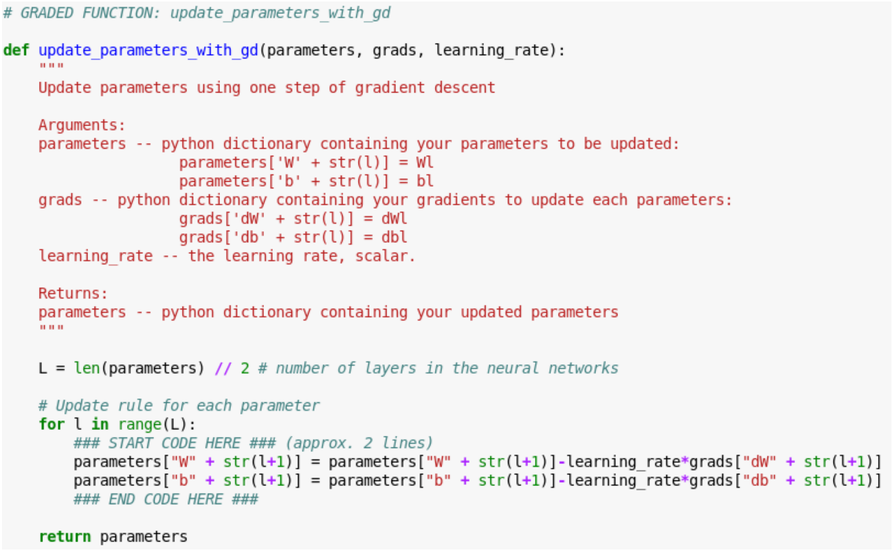
2.1 Optimization

2.1.1 Gradient Descent

梯度下降是每次处理完m个样本后对参数进行一次更新操作,也叫做 Batch Gradient Descent。对 于L层模型,梯度下降法对于各层参数的更新:l=1,...L。

$$W^{[l]} = W^{[l]} - \alpha\ dW^{[l]} \tag{1}$$

$$b^{[l]} = b^{[l]} - \alpha\ db^{[l]} \tag{2}$$



```
# GRADED FUNCTION: update_parameters_with_gd

def update_parameters_with_gd(parameters, grads, learning_rate):
    """
    Update parameters using one step of gradient descent

    Arguments:
    parameters -- python dictionary containing your parameters to be updated:
                    parameters['W' + str(l)] = Wl
                    parameters['b' + str(l)] = bl
    grads -- python dictionary containing your gradients to update each parameters:
                    grads['dW' + str(l)] = dWl
                    grads['db' + str(l)] = dbl
    learning_rate -- the learning rate, scalar.

    Returns:
    parameters -- python dictionary containing your updated parameters
    """

    L = len(parameters) // 2 # number of layers in the neural networks

    # 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
```

图 18: 梯度下降算法

批次梯度下降和随机梯度下降的区别:

- (Batch) Gradient Descent:

```
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    # Forward propagation
    a, caches = forward_propagation(X, parameters)
    # Compute cost.
    cost = compute_cost(a, Y)
    # Backward propagation.
    grads = backward_propagation(a, caches, parameters)
    # Update parameters.
    parameters = update_parameters(parameters, grads)
```

- Stochastic Gradient Descent:

```
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    for j in range(0, m):
        # Forward propagation
        a, caches = forward_propagation(X[:,j], parameters)
        # Compute cost
        cost = compute_cost(a, Y[:,j])
        # Backward propagation
        grads = backward_propagation(a, caches, parameters)
        # Update parameters.
        parameters = update_parameters(parameters, grads)
```

Stochastic Gradient Descent (SGD) 随机梯度下降法。等同于 mini-batch 中每个 mini-batch 只有一个样本的梯度下降法。此时,梯度下降的更新法则就变成每个样本都要计算一次,而不是此前的对整个样本集计算一次。

实现随机梯度下降需要三个循环：

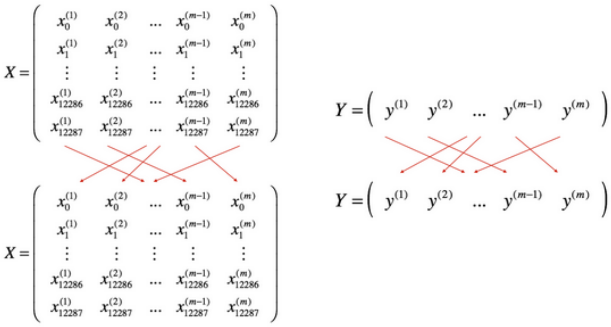
- 最外层的迭代次数
- m个训练样本
- 每一层参数的更新，from $(W^{[1]}, b^{[1]})$ to $(W^{[L]}, b^{[L]})$

Remember:

- The difference between gradient descent, mini-batch gradient descent and stochastic gradient descent is the number of examples you use to perform one update step.
- You have to tune a learning rate hyperparameter α.
- With a well-turned mini-batch size, usually it outperforms either gradient descent or stochastic gradient descent (particularly when the training set is large).

2.1.2 Mini-Batch Gradient Descent

随机播放：创建一个洗牌版本的训练集（X，Y），如下所示。 X和Y的每列表示训练示例,X和Y之间的随机混洗是同步进行的。这样，在混洗后，X的第i列就是Y中的第i个标签对应的例子。洗牌步骤确保了这些例子将被随机分解成不同的小批量。



$X = \begin{pmatrix} x_0^{(1)} & x_0^{(2)} & \dots & x_0^{(m-1)} & x_0^{(m)} \\ x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(m-1)} & x_1^{(m)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{12286}^{(1)} & x_{12286}^{(2)} & \dots & x_{12286}^{(m-1)} & x_{12286}^{(m)} \\ x_{12287}^{(1)} & x_{12287}^{(2)} & \dots & x_{12287}^{(m-1)} & x_{12287}^{(m)} \end{pmatrix}$
$Y = \begin{pmatrix} y^{(1)} & y^{(2)} & \dots & y^{(m-1)} & y^{(m)} \end{pmatrix}$

图 19: mini_batch

分区：将洗牌（X，Y）分成mini_batch_size（这里64）的小批量。 请注意，训练示例的数量并不总是被mini_batch_size整除。 最后一个小批量可能会更小，但不需要担心这一点。当最终的mini_batch小于完整的mini_batch_size时，它将如下所示：

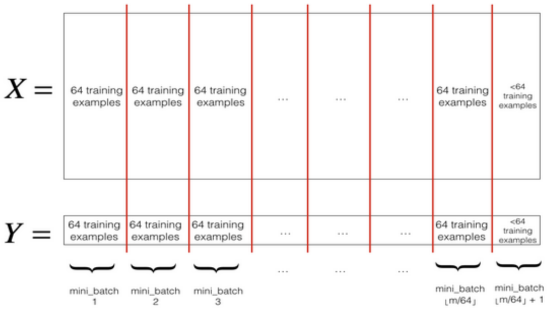
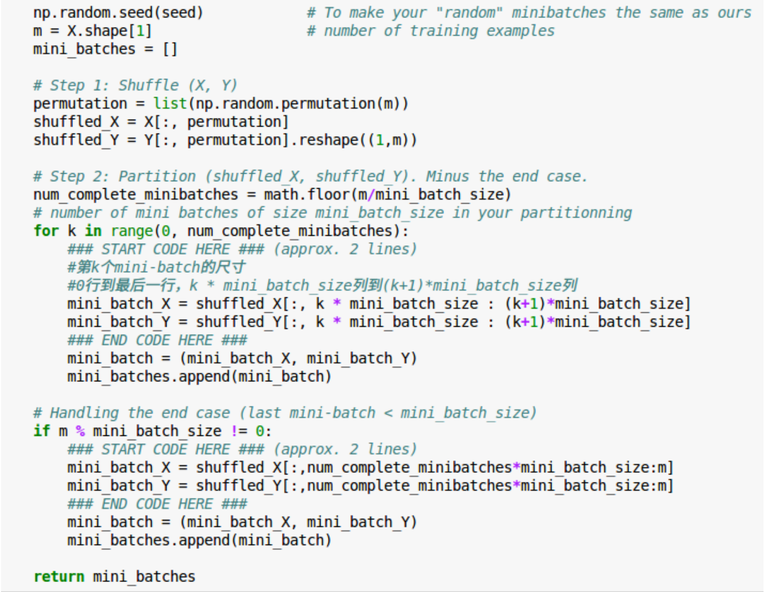


图 20: mini_batch



```
    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))

    # 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 your partitionning
    for k in range(0, num_complete_minibatches):
        ### START CODE HERE ### (approx. 2 lines)
        #第k个mini-batch的尺寸
        #0行到最后一行, k * mini_batch_size列到(k+1)*mini_batch_size列
        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:m]
        mini_batch_Y = shuffled_Y[:,num_complete_minibatches*mini_batch_size:m]
        ### END CODE HERE ###
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    return mini_batches
```

图 21: mini-batch梯度下降算法

2.1.3 Momentum

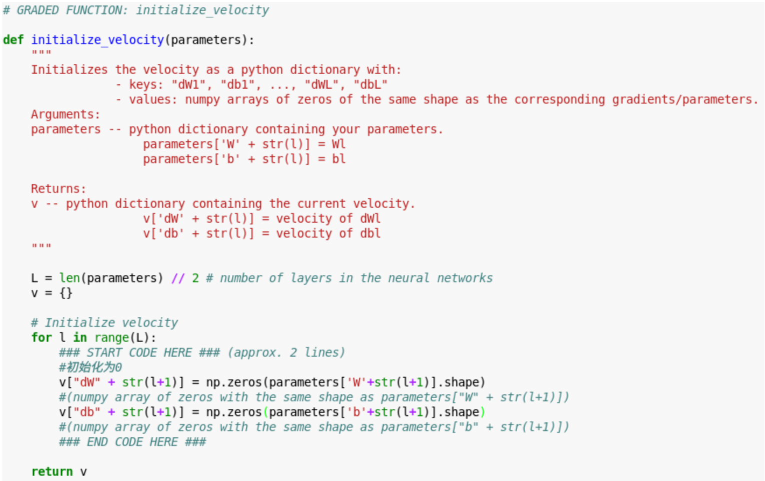
min-batch 梯度下降是在看过训练集的一部分子数据集之后,就开始了参数的更新,会在参数更新过程中出现偏差震荡。采用动量梯度下降法可以减缓震荡的出现。momentum 方式是在参 数更新时候,参考历史的参数值,以平滑参数的更新。

动量考虑过去的渐变以平滑更新。我们将把先前梯度的“方向”存储在变量ν中。这将是以前步骤中梯度的指数加权平均值。也可以认为ν是一个滚动下坡的球的“速度”,根据山坡的坡度/坡度的方向建立速度（和动量）。

使梯度下降在纵轴的方向摆动小了，横轴的方向运动更快，加快优化。

velocity值初始化:initialize_velocity 在Python中是一个字典,初始为0矩阵,尺寸与 grads 一致,l=1,....,L

```
v["dW" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["W" + str(l+1)])
v["db" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["b" + str(l+1)])
```



```
# GRADED FUNCTION: initialize_velocity

def initialize_velocity(parameters):
    """
    Initializes the velocity as a python dictionary with:
                - keys: "dW1", "db1", ..., "dWL", "dbL"
                - values: numpy arrays of zeros of the same shape as the corresponding gradients/parameters.
    Arguments:
    parameters -- python dictionary containing your parameters.
                    parameters['W' + str(l)] = Wl
                    parameters['b' + str(l)] = bl

    Returns:
    v -- python dictionary containing the current velocity.
                    v['dW' + str(l)] = velocity of dWl
                    v['db' + str(l)] = velocity of dbl
    """

    L = len(parameters) // 2 # number of layers in the neural networks
    v = {}

    # Initialize velocity
    for l in range(L):
        ### START CODE HERE ### (approx. 2 lines)
        #初始化为0
        v["dW" + str(l+1)] = np.zeros(parameters['W'+str(l+1)].shape)
        #(numpy array of zeros with the same shape as parameters["W" + str(l+1)])
        v["db" + str(l+1)] = np.zeros(parameters['b'+str(l+1)].shape)
        #(numpy array of zeros with the same shape as parameters["b" + str(l+1)])
        ### END CODE HERE ###

    return v
```

图 22: velocity值初始化

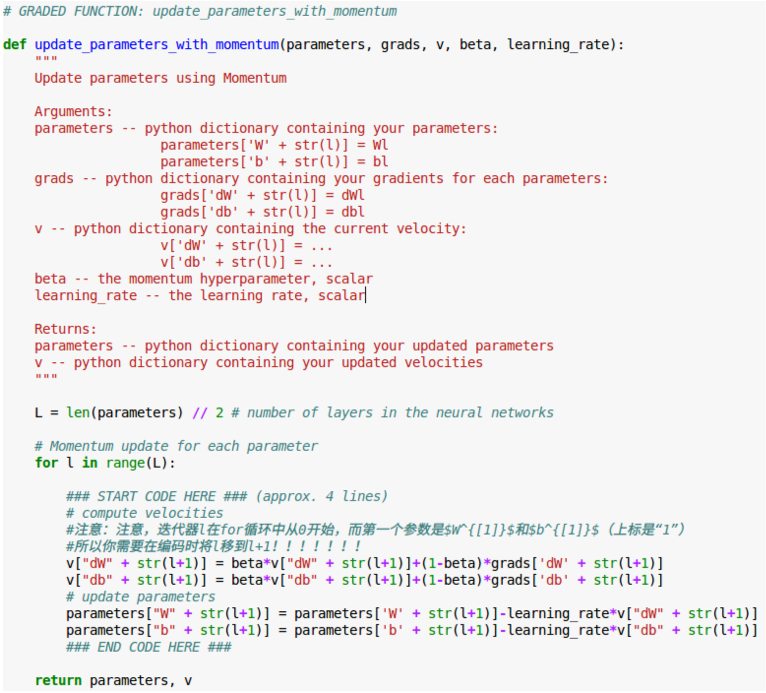
Implement the parameters update with momentum. The momentum update rule is, for l=1,...,L:

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

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

其中L是层数，β是动量，α是学习率。所有参数应存储在参数字典中。

注意，迭代器l在for循环中从0开始，而第一个参数是$W^{[1]}$和$b^{[1]}$（上标是“1”）。所以你需要在编码时将l移到l+1。



```python
# GRADED FUNCTION: update_parameters_with_momentum

def update_parameters_with_momentum(parameters, grads, v, beta, learning_rate):
    """
    Update parameters using Momentum

    Arguments:
    parameters -- python dictionary containing your parameters:
                    parameters['W' + str(l)] = Wl
                    parameters['b' + str(l)] = bl
    grads -- python dictionary containing your gradients for each parameters:
                    grads['dW' + str(l)] = dWl
                    grads['db' + str(l)] = dbl
    v -- python dictionary containing the current velocity:
                    v['dW' + str(l)] = ...
                    v['db' + str(l)] = ...
    beta -- the momentum hyperparameter, scalar
    learning_rate -- the learning rate, scalar

    Returns:
    parameters -- python dictionary containing your updated parameters
    v -- python dictionary containing your updated velocities
    """

    L = len(parameters) // 2 # number of layers in the neural networks

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

        ### START CODE HERE ### (approx. 4 lines)
        # compute velocities
        #注意：注意，迭代器l在for循环中从0开始，而第一个参数是$W^{[1]}$和$b^{[1]}$（上标是“1”）
        #所以你需要在编码时将l移到l+1！！！！！！！
        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
```

图 23: Momentum参数更新

2.1.4 Adam

Adam is one of the most effective optimization algorithms for training neural networks. It combines ideas from RMSProp (described in lecture) and Momentum.

1.计算过去梯度的指数加权平均值，并将其存储在变量vv（偏差校正之前）和校正校正（偏置校正）中。

2.计算过去梯度的平方的指数加权平均值，并将其存储在变量ss（偏差校正之前）和校正校正（偏置校正）之间。

3.基于从“1”和“2”的组合信息在方向上更新参数。

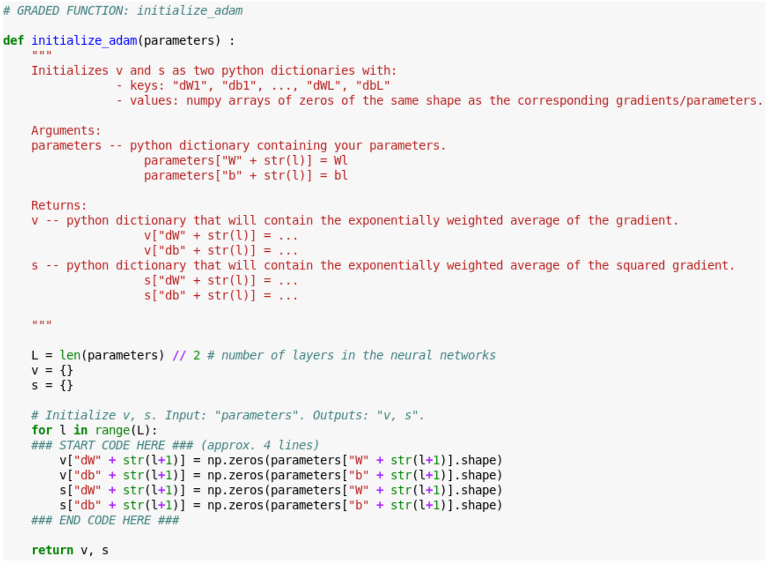
$$
\begin{cases}
v_{dW^{[l]}} = \beta_1 v_{dW^{[l]}} + (1 - \beta_1)\frac{\partial \mathcal{J}}{\partial W^{[l]}} \\
v^{corrected}_{dW^{[l]}} = \frac{v_{dW^{[l]}}}{1-(\beta_1)^t} \\
s_{dW^{[l]}} = \beta_2 s_{dW^{[l]}} + (1 - \beta_2)(\frac{\partial \mathcal{J}}{\partial W^{[l]}})^2 \\
s^{corrected}_{dW^{[l]}}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

图 24: Adam规则

- t counts the number of steps taken of Adam
- L is the number of layers
- β_1 and β_2 are hyperparameters that control the two exponentially weighted averages.
- α is the learning rate
- ε is a very small number to avoid dividing by zero

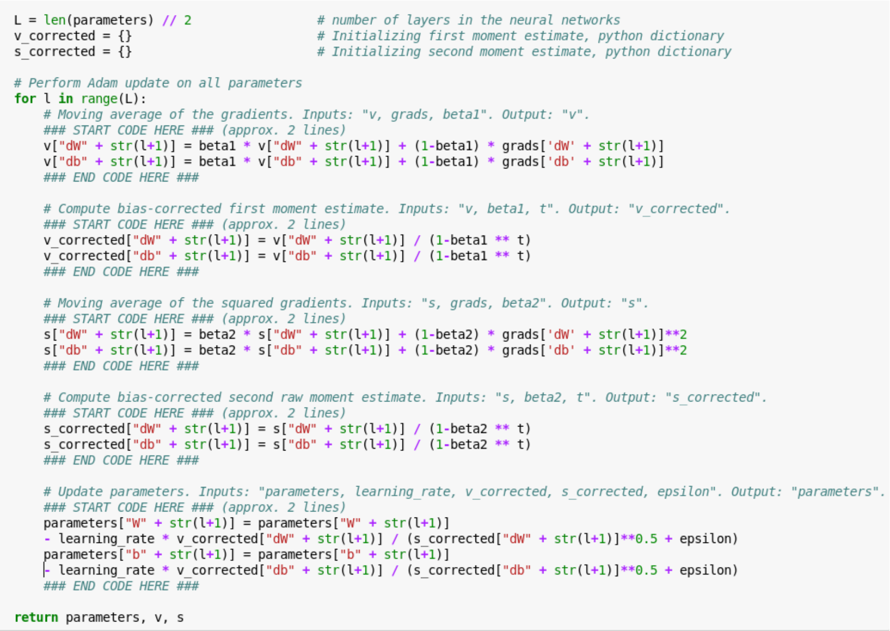
The variables v,s are python dictionaries that need to be initialized with arrays of zeros. Their keys are the same as for grads, that is: for l=1,...,L:

```
v["dW" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["W" + str(l+1)])
v["db" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["b" + str(l+1)])
s["dW" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["W" + str(l+1)])
s["db" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["b" + str(l+1)])
```



```
# GRADED FUNCTION: initialize_adam

def initialize_adam(parameters) :
    """
    Initializes v and s as two python dictionaries with:
                - keys: "dW1", "db1", ..., "dWL", "dbL"
                - values: numpy arrays of zeros of the same shape as the corresponding gradients/parameters.

    Arguments:
    parameters -- python dictionary containing your parameters.
                    parameters["W" + str(l)] = Wl
                    parameters["b" + str(l)] = bl

    Returns:
    v -- python dictionary that will contain the exponentially weighted average of the gradient.
                    v["dW" + str(l)] = ...
                    v["db" + str(l)] = ...
    s -- python dictionary that will contain the exponentially weighted average of the squared gradient.
                    s["dW" + str(l)] = ...
                    s["db" + str(l)] = ...

    """

    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):
    ### START CODE HERE ### (approx. 4 lines)
        v["dW" + str(l+1)] = np.zeros(parameters["W" + str(l+1)].shape)
        v["db" + str(l+1)] = np.zeros(parameters["b" + str(l+1)].shape)
        s["dW" + str(l+1)] = np.zeros(parameters["W" + str(l+1)].shape)
        s["db" + str(l+1)] = np.zeros(parameters["b" + str(l+1)].shape)
    ### END CODE HERE ###

    return v, s
```

图 25: Adam参数初始化



```
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):
    # 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-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)] = parameters["W" + str(l+1)]
    - learning_rate * v_corrected["dW" + str(l+1)] / (s_corrected["dW" + str(l+1)]**0.5 + epsilon)
    parameters["b" + str(l+1)] = parameters["b" + str(l+1)]
    - learning_rate * v_corrected["db" + str(l+1)] / (s_corrected["db" + str(l+1)]**0.5 + epsilon)
    ### END CODE HERE ###

return parameters, v, s
```

图 26: Adam算法

总结： We have already implemented a 3-layer neural network. You will train it with:

- Mini-batch Gradient Descent: it will call your function:

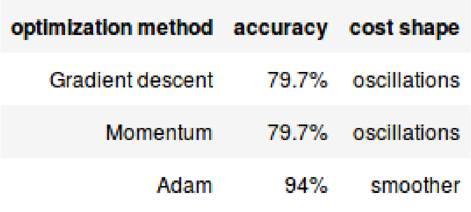
  update_parameters_with_gd()

- Mini-batch Momentum: it will call your functions:

  initialize_velocity() and update_parameters_with_momentum()

- Mini-batch Adam: it will call your functions:

  initialize_adam() and update_parameters_with_adam()



optimization method	accuracy	cost shape
Gradient descent	79.7%	oscillations
Momentum	79.7%	oscillations
Adam	94%	smoother

图 27: 三种算法的对比

3 Programming Frameworks

3.1 Tensorflow