Deep Learning

卢婧宇

2017 年 10 月 21 日

目录

1 Neural Networks & Deep Learning 神经网络和深度学习

1.1 Building your Deep Neural Network: Step by Step(第四周作业)

1.1.1 Outline of the Assignment

- 双隐藏层和 L 层神经网络模型的参数初始化
- 做前向传播
  - 计算正向传播的 LINEAR 部分，线性部分即 Z=WX+b 这部分，输出部分就是 A，就是将线性部分的结果输入到激活函数所产生的结果。
  - 采用 RELU 或者 sigmoid 激活函数计算结果值
  - 联合上述两个步骤，进行前向传播操作 [LINEAR->ACTIVATION]
  - 对输出层之前的 L-1 层，做 L-1 次的前向传播 [LINEAR->RELU] ，并将结果输出到第 L 层 [LINEAR->SIGMOID]。所以在前面 L-1 层的激活函数是 RELU，在输出层的激活函数是 sigmoid。
- 计算损失函数
- 做后向传播操作
  - 计算神经网络反向传播的 LINEAR 部分
  - 计算激活函数（RELU 或者 sigmoid）的梯度
  - 结合前面两个步骤，产生一个新的后向函数 [LINEAR->ACTIVATION]
- 更新参数

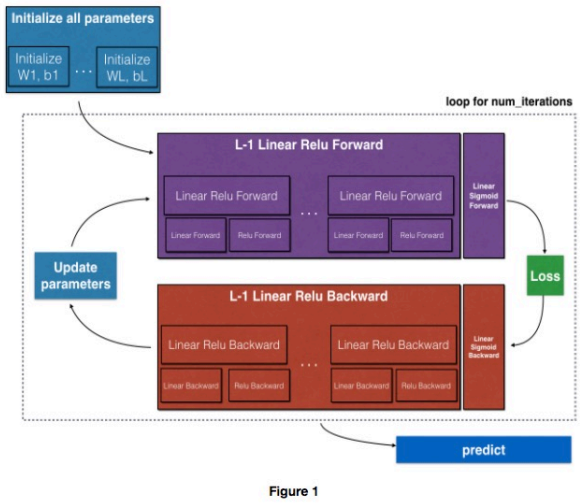

Figure 1

图 1: 流程图

1.1.2 L-layer Neural Network

对于 L 层模型：

- 模型结构：[LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID。所以 L−1 层是需要用到 ReLU 激活函数的。输出层用的是 sigmoid 函数。
- 权重矩阵采用仍旧是随机化初始化的方式：np.random.rand(shape) * 0.01
- 偏移矩阵仍旧是 0 矩阵进行处初始化：np.zeros(shape).
- 我们将每层的神经元数量 n[l] 信息进行存储，layer_dims。例如在平面数据分类模型中 layer_dims 的值是 [2,4,1]，其中输入层的神经元个数是 2，隐藏层的神经元个数是 4，输出层的神经元个数是 1。对应的 W1 尺寸 = (4,2), b1 尺寸 = (4,1), W2 尺寸 = (1,4) ， b2 尺寸 = (1,1)。

GRADED FUNCTION: initialize_parameters_deep

```
for l in range(1, L):
    ### START CODE HERE ### (≈ 2 lines of code)
    parameters['W' + str(l)] = np.random.randn(layer_dims[l],layer_dims[l-1])*0.01
    parameters['b' + str(l)] = np.zeros((layer_dims[l],1))
    ### END CODE HERE ###
```

图 2: 初始化 W，b

1.1.3 Forward propagation module

线性传播部分：
GRADED FUNCTION: linear_forward

```
### START CODE HERE ### (≈ 1 line of code)
Z = np.dot(W,A)+b
### END CODE HERE ###
```

图 3: Z 的计算

激活部分:
运用两个激活函数：
Sigmoid. 在这个步骤我们需要两个结果，一个是激活函数的结果值，另一个是包含 “Z” 的 “cache” 值，这个我们在后向传播过程需要用到。A, activation_cache = sigmoid(Z)
ReLU.A=RELU(Z)=max(0,Z). 同样结果值有两部分，其一是激活函数结果值 “A” ，另一个是包含 “Z” 的 “cache” 值。A, activation_cache = relu(Z)
(a) 相邻两层的激活实现
GRADED FUNCTION: linear_activation_forward

```
if activation == "sigmoid":
    # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
    ### START CODE HERE ### (≈ 2 lines of code)
    Z, linear_cache = linear_forward(A_prev,W,b)
    A, activation_cache = sigmoid(Z)
    ### END CODE HERE ###

elif activation == "relu":
    # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
    ### START CODE HERE ### (≈ 2 lines of code)
    Z, linear_cache = linear_forward(A_prev,W,b)
    A, activation_cache = relu(Z)
    ### END CODE HERE ###
```

图 4: 激活函数

(b)L-Layer Model (L 层模型)

那么对于 L 层的神经网络，激活函数为 RELU 的 linear_activation_forward 需要重复 L-1 次，而最后的输出层采用的参数为 SIGMOID 的 linear_activation_forward 。

GRADED FUNCTION: L_model_forward

[LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID model

```
# Implement [LINEAR -> RELU]*(L-1). Add "cache" to the "caches" list.
for l in range(1, L):
    A_prev = A
    ### START CODE HERE ### (≈ 2 lines of code)
    A, cache = linear_activation_forward(A_prev,parameters['W' + str(l)],parameters['b'+str(l)],activation = "relu")
    caches.append(cache)
    ### END CODE HERE ###

# Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
### START CODE HERE ### (≈ 2 lines of code)
AL, cache = linear_activation_forward(A,parameters['W'+str(L)],parameters['b'+str(L)],activation = "sigmoid")
caches.append(cache)
### END CODE HERE ###

assert(AL.shape == (1,X.shape[1]))

return AL, caches
```

图 5: L 层前向传播

1.1.4 Cost function

```
m = Y.shape[1]

# Compute loss from aL and y.
### START CODE HERE ### (≈ 1 lines of code)
cost = -(1/m)*np.sum(np.multiply(Y,np.log(AL))+np.multiply((1-Y),np.log(1-AL)))
### END CODE HERE ###

cost = np.squeeze(cost)      # To make sure your cost's shape is what we expect (e.g. this turns [[17]] into 17).
assert(cost.shape == ())

return cost
```

图 6: 代价函数

1.1.5 Backward propagation module

后向传播是为了计算各个参数梯度，其模型如下：

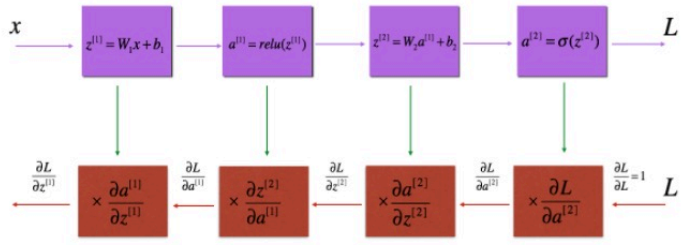

Figure 3 : Forward and Backward propagation for *LINEAR->RELU->LINEAR->SIGMOID*
The purple blocks represent the forward propagation, and the red blocks represent the backward propagation.

图 7: 后向传播

和之前的前向传播类似，后向传播模块的建立分以下三个步骤：

- 后向 LINEAR（Linear backward）
- ReLU 或者 sigmoid 激活函数的后向 LINEAR -> ACTIVATION
- [LINEAR -> RELU]× (L-1) -> LINEAR -> SIGMOID backward (whole model)

后向 Linear
三个输出 $(dW^{[l]}, db^{[l]}, dA^{[l]})$ 可以通过输入 $dZ^{[l]}$ 计算获得。公式如下:

$$dW^{[l]} = \frac{\partial \mathcal{L}}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$$

$$db^{[l]} = \frac{\partial \mathcal{L}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)}$$

$$dA^{[l-1]} = \frac{\partial \mathcal{L}}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]}$$

图 8: 后向传播公式

GRADED FUNCTION: linear_backward

```
A_prev, W, b = cache
m = A_prev.shape[1]

### START CODE HERE ### (≈ 3 lines of code)
dW = (1/m)*np.dot(dZ,A_prev.T)
db = (1/m)*np.sum(dZ,axis=1,keepdims=True)
dA_prev = np.dot(W.T,dZ)
### END CODE HERE ###

assert (dA_prev.shape == A_prev.shape)
assert (dW.shape == W.shape)
assert (db.shape == b.shape)

return dA_prev, dW, db
```

图 9: 后向传播公式

(a)Linear-Activation backward

对于 sigmoid 函数，可以定义两个函数：

sigmoid_backward: 用以计算 SIGMOID 单元：dZ = sigmoid_backward(dA, activation_cache) 其用到的 cache 值是 Z 值

relu_backward: 用以计算 RELU 的 backward propagation：dZ = relu_backward(dA, activation_cache)

对于 g(.) 的激活函数：

sigmoid_backward 和 relu_backward 的计算如下:$dZ^{[l]} = dA^{[l]} * g'(Z^{[l]})$

GRADED FUNCTION: linear_activation_backward

```
linear_cache, activation_cache = cache

if activation == "relu":
    ### START CODE HERE ### (≈ 2 lines of code)
    dZ = relu_backward(dA, activation_cache)
    dA_prev, dW, db = linear_backward(dZ, linear_cache)
    ### END CODE HERE ###

elif activation == "sigmoid":
    ### START CODE HERE ### (≈ 2 lines of code)
    dZ = sigmoid_backward(dA, activation_cache)
    dA_prev, dW, db = linear_backward(dZ, linear_cache)
    ### END CODE HERE ###

return dA_prev, dW, db
```

图 10: 线性激活函数反向传播

(b)L-Model Backward

对整个神经网络做后向传播，定义函数为 L_model_forward。在每次的迭代过程中，我们都将 cache 值 =(X,W,b, z) 保留，用以后向模块中梯度的计算。在 L_model_forward 中，我们是重复了 L 次上述的步骤。

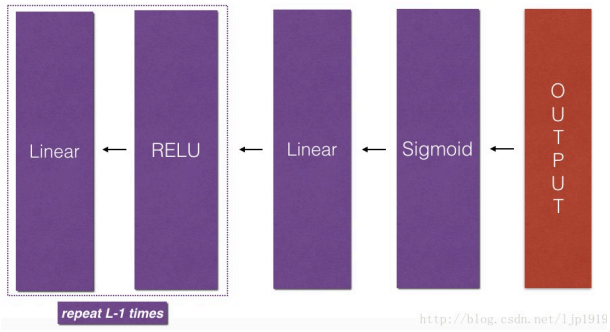

图 11: L 层反向传播

后向传播初始化：

对于后向传播，我们知道前向传播的输出是 $A^{[L]} = \sigma(Z^{[L]})$，我们需要计算 $dAL = \frac{\partial L}{\partial A^{[L]}}$，我们用以下的公式表示：dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL)) derivative of cost with respect to AL

之后，我们可以用这个后向激活的梯度 dAL 进行向后传播。再从 dAL 计算 LINEAR->SIGMOID 的后向传播结果。对于 LINEAR->RELU backward 函数，我们可以采用 for 循环来处理这 L-1 次操作。在此期间，我们要存储 dA, dW, db，本文用 grads 字典来存储：$grads[\text{“}dW'' + str(l)] = dW^{[l]}$

例如, 对于 l=3，则 $dW^{[l]}$ 以 grads[“dW3”] 形式存储。

模型如下：[LINEAR->RELU] × (L-1) -> LINEAR -> SIGMOID
GRADED FUNCTION: L_model_backward

```
# Initializing the backpropagation
### START CODE HERE ### (1 line of code)
dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))
### END CODE HERE ###

# Lth layer (SIGMOID -> LINEAR) gradients. Inputs: "AL, Y, caches". Outputs: "grads["dAL"], grads["dWL"], grads["dl
### START CODE HERE ### (approx. 2 lines)
current_cache = caches[L-1]
grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] = linear_activation_backward(dAL, current_cache, a
### END CODE HERE ###

for l in reversed(range(L-1)):
    # lth layer: (RELU -> LINEAR) gradients.
    # Inputs: "grads["dA" + str(l + 2)], caches". Outputs: "grads["dA" + str(l + 1)] , grads["dW" + str(l + 1)] , g
    ### START CODE HERE ### (approx. 5 lines)
    current_cache = caches[l]
    dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["dA" + str(l + 2)], current_cache, activatio
    grads["dA" + str(l + 1)] =dA_prev_temp
    grads["dW" + str(l + 1)] = dW_temp
    grads["db" + str(l + 1)] = db_temp
    ### END CODE HERE ###

return grads
```

图 12: L 层反向传播模型

Update Parameters
采用梯度下降进行参数的更新：$W^{[l]} = W^{[l]} - \alpha dW^{[l]}$ $b^{[l]} = b^{[l]} - \alpha db^{[l]}$

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

# Update rule for each parameter. Use a for loop.
### START CODE HERE ### (≈ 3 lines of code)
for l in range(L):
    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 13: 更新

1.1.6 General methodogy 通用模型

As usual you will follow the Deep Learning methodology to build the model:

- Initialize parameters / Define hyperparameters
- Loop for num_iterations:
  - Forward propagation
  - Compute cost function
  - Backward propagation
  - Update parameters (using parameters, and grads from backprop)
- Use trained parameters to predict labels

lement those two models!

2 Improve Deep Neural Networks

2.1 Initialization

一个好的初始化参数能够加速梯度下降的收敛，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。

2.1.1 Zero initialization 全零初始化

parameters = initialize_parameters_zeros(layers_dims)

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

图 14: 全零初始化

效果很差，整个迭代过程损失函数并没有下降。该参数下的模型对于训练集和测试集的预测结果全为 0。

该模型预测结果是没有边界的，因为都是预测结果都是一类，都是 0。这种全 0 的参数初始化方式就无法打破网络的对称。这意味着每层中的每个神经元学习内容都是相同的。当每层神经元个数 =1，即 $n^{[l]} = 1$，那么该神经网络的性能便退化成线性分类器，比如逻辑回归。

所以，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。

2.1.2 Random initialization 随机初始化

当权重矩阵随机初始化后，每个神经元学习将从不同输入函数学习到不同东西。下面我们随机初始化权重矩阵，但是初始化值设置为较大，下面代码中我们乘以 10。

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.

```
np.random.seed(3)               # This seed makes sure your "random" numbers will be the as
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 15: 随机初始化

分析：

起始的代价函数很大，这是由于我们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 $log(a^{[3]}) = log(0)$，其代价是无穷大的。

过大或者过小的权重矩阵将导致梯度的暴涨或者快速衰减，这都会减缓优化算法获取最优结果的速度。为此，我们需要控制随机化的权重矩阵大小。

2.1.3 He initialization He 初始化

He 初始化方式（He et al., 2015.）正是为解决上面问题而提出的。这种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]))。另一种相识的初始化方式 Xavier 方式，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 16: He 初始化

总结:

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 17: 总结

2.2 Regularization 正则化

对于 L2 正则化模型，需要用到的参数是 lambd 值。该正则化模型需要用到 compute_cost_with_regularization() 和 backward_propagation_with_regularization()。

而对于 dropout 模型则需要设置 keep_prob 值作为神经元节点的概率开关。需要用到的函数是 forward_propagation_with_dropout() 和 backward_propagation_with_dropout()。

```
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
                                        # but this assignment will only explore one at a t
    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)
```

图 18: 正则化

存在过拟合。

2.2.1 L2 Regularization

公式:

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

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

图 19: 表达式

定义 compute_cost_with_regularization() 函数表示公式 (2) 的代价函数。$\sum_k \sum_j W_{k,j}^{[l]2}$ 的计算，我们可以采用:np.sum(np.square(Wl))

由于我们有三个 W 参数 $W^{[1]}, W^{[2]} and W^{[3]}$, 所以需要将其分别计算，求和，再乘以 $(1/m) * (\lambda/2)$。

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

cost = cross_entropy_cost + L2_regularization_cost
```

图 20: L2

由于我们修改了代价函数，所以后向传播也需要做修改。同理，我们也仅仅需要修改 dW1, dW2 and dW3。对于这三者，每一项都需要加入正则项的梯度 $\frac{d}{dW}(\frac{1}{2}\frac{\lambda}{m}W^2) = \frac{\lambda}{m}W$。

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

dZ3 = A3 - Y

### START CODE HERE ### (approx. 1 line)
dW3 = 1./m * np.dot(dZ3, A2.T) + lambd/m*W3
### 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/m*W2
### 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/m*W1
### 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
```

图 21: L2

设置 λ 的值: parameters = model(train_X, train_Y, lambd = 0.7)

讨论:

λ 的值是可以通过对 dev set 的调整获取最优值。L2 正则使得边界更加平滑，但是如果 λ 过大，会产生过度平滑，退化成一个线性回归，从而带来高偏差问题。L2 正则化是基于较小的权值比较大的权值模型更简单这一假设。通过正则化，我们在代价函数中引入权重的平方值，这导致在反向传播时候，权重变得更小。

2.2.2 Dropout Regularization

dropout 正则的前向传播:

$d^{[1]}$ 和 $a^{[1]}$ 尺寸一致,所以 np.random.rand() 进行向量的初始化。对于矩阵 $D^{[1]} = [d^{[1](1)}d^{[1](2)}...d^{[1](m)}]$ 尺寸和 $A^{[1]}$ 一致，初始化方式类似。

根据 keep_prob 对 $D^{[1]}$ 矩阵做 0-1 划分。方式如 X = (X < 0.5)，其实这是产生一个 0-1 矩阵 mask，用于神经元节点的筛选。$A^{[1]}$ to $A^{[1]}*D^{[1]}$ 进行神经元节点的筛选操作。$A^{[1]}$ /keep_prob，使得前后的期望值一致（inverted dropout）

```
# 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-
D1 = np.random.rand(A1.shape[0],A1.shape[1])                                        # Step 1:
D1 = (D1<keep_prob)                                    # Step 2: convert entries of D1 to 0 or
A1 = A1*D1                                          # Step 3: shut down some neurons of A1
A1 = A1/keep_prob                                          # Step 4: scale the value of neurons
### 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:
D2 = (D2<keep_prob)                                        # Step 2: convert entries of D2 to 0
A2 = A2*D2                                          # Step 3: shut down some neurons of A2
A2 = A2/keep_prob                                          # Step 4: scale the value of neurons
### 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 22: dropout 正则化前向传播

dropout 正则的后向传播:

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。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 进行一定程度放大，则其导数 $A^{[1]}$ 也需要做同比例的操作。

```
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
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
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)
```

图 23: dropout 正则化后向传播

讨论：

一个常见的错误是，将 dropout 同时用于训练集和测试集，记住一点：dropout 仅仅用于训练集！

结论：

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 24: 结论

2.3 Gradient Checking

在神经网络计算过程中，对后向传播的梯度进行校验，确保其计算无误。至于，前向传播，由于相对简单，所以，一般不会出错，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。

2.3.1 1-dimensional gradient checking 一维梯度校验

计算梯度近似值 “gradapprox”: $gradapprox = \frac{J^+ - J^-}{2\xi}$

计算两者 difference:$difference = \frac{||grad - gradapprox||_2}{||grad||_2 + ||gradapprox||_2}$

范数的计算可以用 np.linalg.norm(...)。当 difference 足够小 ($< 10^{-7}$)，则可以视为梯度校验通过。

```
# Compute gradapprox using left side of formula (1). epsilon is small enough, you don't need
### START CODE HERE ### (approx. 5 lines)
thetaplus = theta+epsilon                                  # Step 1
thetaminus = theta-epsilon                                 # Step 2
J_plus = forward_propagation(thetaplus,x)                                  # Step 3
J_minus = forward_propagation(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)                                    #
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 25: 一维梯度校验

2.3.2 N-dimensional gradient checking n 维梯度校验

所以，要注意看 difference 值是否和阈值相距很大。本文为何就差那么一些，导致需要修改 epsilon，可能是由于代价函数在局部存在毛刺，导致估算值和后向梯度计算结果，存在超于阈值的偏差。另外，relu

```
# 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
    ### 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))
    ### 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(thetaplus))
    ### 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)
difference = numerator/denominator                                  # Step 3'
### END CODE HERE ###
  #注意这里的epsilon值的问题，如果1e-6是可以的，是的梯度校验通过，epsilon值越小，反而和导数越不一致
if difference > 1e-7:
    print ("\033[93m" + "There is a mistake in the backward propagation! difference = " + str(di
else:
    print ("\033[92m" + "Your backward propagation works perfectly fine! difference = " + str(di

return difference
```

图 26: n 维梯度校验

的导数在 0 处有歧义，也可能导致此处的不够准确。

2.4 Optimization Methods

2.4.1 Gradient Descent 梯度下降法

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。

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

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

update_parameters_with_gd:

```
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 27: Gradient Descent

Stochastic Gradient Descent (SGD) 随机梯度下降法。这等同于 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。

- **(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)
```

图 28: SGD vs GD

当训练集很大时，这种方法可以明显提高运行速度，但是参数会沿着最小方向震荡，而不是平滑地收敛。

注意 SGD 共需要三个循环:

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

谨记:

- gradient descent, mini-batch gradient descent 和 stochastic gradient descent 之间的区别在于梯度更新所用到的样本数据量。
- 超参数学习率　是需要调整获取到
- mini-batch 的尺寸也是调整获取到的，所以也是一个超参数。一般情况下这种方式比另外两者更好，特别是当训练集特别大的时候。

2.4.2 Mini-Batch 梯度下降

There are two steps:

- Shuffle（洗牌）X 和 Y 的随机要一致，否则 Y 值不能与 X 匹配

- Partition(分割) 尾部的数据可能小于一个 mini_batch_size，所以对于最后一个 mini-batch 要注意处理。

当样本数无法被 mini_batch_size 整除的时候，最后一个 mini-batch<mini_batch_size=64。[s] 表示 s 向下取整（Python 中实现: math.floor(s))。

最后一个 min-batch 中样本数量 = (m−mini_batch_size× m/mini_batch_size)。

```
# 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_batc
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] #[取整行: , 取列]
    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
```

图 29: Mini-Batch

一般 mini-batch size 的取值是 2^n，如 16, 32, 64, 128 等。

2.4.3 Momentum 动量梯度下降法

由于 min-batch 梯度下降法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，那么就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。

velocity 值初始化：initialize_velocity 在 Python 中是一个字典，初始为 0 矩阵，其尺寸与 grads 一致。

带 momentum 的参数更新，l 从 1 开始。:

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

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

图 30: 参数更新

update_parameters_with_momentum

注意:

velocity 初始化为 zeros，所以算法需要迭代一定次数以建立起速度，实现每次迭代的 bigger steps。当 $\beta = 0$，则退化成标准的梯度下降法。

β 值的选取：

越大，历史梯度值引入到当前值的权重越大，更新就会越平滑。但是如果太大，则会导致更新平滑过度。一般取值在 0.8 到 0.999 之间，常取 0.9。可以通过尝试几个 β 值，然后看哪个值在降低 cost function J 效果最好，来获取最优值。

2.4.4 Adam 算法

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

图 31: 公式

过程

```
# 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-np.power(beta1,2))
    v_corrected["db" + str(l+1)] = v["db" + str(l+1)]/(1-np.power(beta1,2))
    ### 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-beta2)
    s_corrected["db" + str(l+1)] = s["db" + str(l+1)]/(1-beta2)
    ### 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_co
    parameters["b" + str(l+1)] = parameters["W" + str(l+1)]-learning_rate*v_corrected["db" + str(l+1)] /np.sqrt(s_co
    ### END CODE HERE ###

return parameters, v, s
```

图 32: 公式

2.4.5 总结

Momentum 一般都是有助于提升速度，但是当学习率较小，数据集相对简单的时候，其性能的优越性没有太明显。我们在优化算法中看到的那些较大的震荡是由于一些 minibatches 相对更加复杂所造成的。

从运行结果可以看出，Adam 算法比 mini-batch gradient descent 和 Momentum 都要显得优越。对于 model 如果在简单数据集上，迭代次数更多的话，这三种优化算法都会产生较好的结果，但是我们也

可以看出，Adam 算法收敛得更快些。

Adam 算法的优点::

内存要求低 (尽管比 gradient descent 和 gradient descent with momentum 要高些) 一般微调超参数就可以获得较好的结果 (除了 α)

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

图 33: 结论