Verificação de Gradiente

Bem vindo a mais uma tarefa do curso de Aprendizado em Profundidade!! Nesta tarefa você irá aprender a implementar e utilizar a verificação de gradiente.

Você é parte de uma equipe reponsável por desenvolver um sistema de pagamento através de celular disponível mundialmente, e foi solicitado que você desenvolvesse um modelo de aprendizado em profundidade para detectar fraudes toda vez que fosse feito um pagamento. Você deve verificar se este pagamento foi fraudulento, por exemplo, verificando se algum hacker entrou na conta daquele cliente.

Acontece que a propagação para trás é desafiadora para ser implementada e, muitas vezes, apresenta problemas. Como esta é uma missão crítica para a aplicação o CEO da sua empresa quer ter certeza que a implementação da propagação para trás está correta. Para provar que o sistema está funcionando você vai utilizar a verificação de gradiente.

Vamos a ela!!

In [1]:
```
# Packages
import numpy as np
from testCases import *
from gc_utils import sigmoid, relu, dictionary_to_vector, vector_to_dict
ionary, gradients_to_vector
```

1) Como funciona a verificação de gradiente?

A propagação para trás computa os valores dos gradientes $\frac{\partial J}{\partial \theta}$, onde θ indica os parâmetros do modelo. J é determinado utilizando a propagação para frente e a função de perda.

Como a propagação para frente é reativamente simples de ser implementada, você tem certeza de que ela está correta e você está confidente de que ela está determinando o valor de J corretamente. Portanto, você pode utilizar o seu código para determinar J e verificar a computação de $\frac{\partial J}{\partial \theta}$.

Vamos olhar novamente a definição de derivada (ou gradiente):

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

Se você não está familiarizado com a notação "$\lim_{\varepsilon \to 0}$", isto é apenas uma forma de dizer que "ε é muito muito pequena."

Nós sabemos que:

- $\frac{\partial J}{\partial \theta}$ é o que você quer verificar.
- Você pode determinar $J(\theta + \varepsilon)$ e $J(\theta - \varepsilon)$ (no caso de θ ser um número real), pois você tem certeza de que a implementação de J está correta.

Vamos utilizar a equação (1) e um valor pequeno para ε para convencer o CEO de que o código está computando $\frac{\partial J}{\partial \theta}$ corretamente!

2) Verificação de gradiente em 1 dimensão

Considere uma função linear em 1D $J(\theta) = \theta x$. O modelo contém apenas uma única variável real como parâmetro θ, e recebe x como entrada.

Você deve implementar o código para determinar $J(.)$ e sua derivada $\frac{\partial J}{\partial \theta}$. Você irá então utilizar a verificação de gradiente para ter certeza que sua implementação da derivada para J está correta.

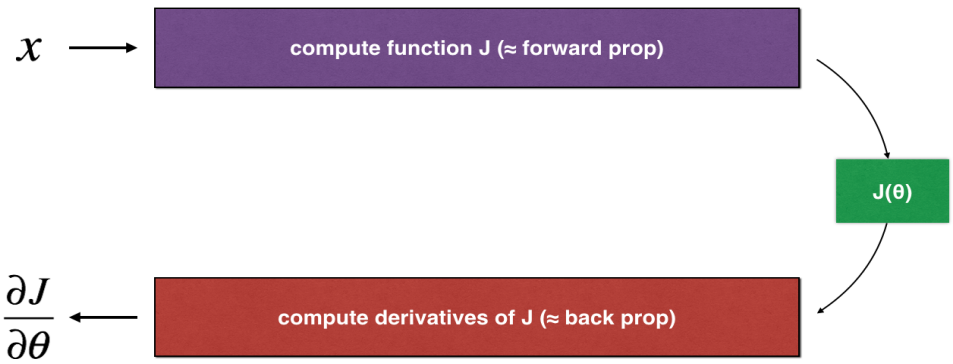

Figura 1 : **modelo linear em 1D**

O diagrama acima mostra as principais etapas de computação: Primeiro comece com x, e determine o valor da função $J(x)$ ("propagação para frente"). Compute então a derivada $\frac{\partial J}{\partial \theta}$ ("propagação para trás").

Exercício: implemente a "propagação para frente" e "propagação para trás" para esta função simples. Isto é , compute os dois $J(.)$ ("propagação para frente") e sua derivada com relação a θ ("propagação para trás"), em duas funções separadas.

In [29]:
```
# FUNÇÃO DE AVALIAÇÃO: propagação para frente

def forward_propagation(x, theta):
    """
    Implemente a propagação para frente (compute J) apresentado na Figur
a 1 (J(theta) = theta * x)

    Argumentos:
    x -- uma entrada com valor real
    theta -- nosso parâmetro, também um número real

    Retorna:
    J -- o valor da função J, determinado utilizando a formula J(theta)
= theta * x
    """

    ### COMECE SEU CÓDIGO AQUI ### (approx. 1 linha)
    J = np.dot(theta, x)
    ### TERMINE O CÓDIGO AQUI ###

    return J
```

In [30]:
```
x, theta = 2, 4
J = forward_propagation(x, theta)
print ("J = " + str(J))
```

```
J = 8
```

Saída esperada:

<table style=> ** J ** 8 </table>

Exercício: Agora, implemente a propagação para trás (computação da derivada) da Figura 1. Isto é, compute a derivada de $J(\theta) = \theta x$ com relação a θ. Para auxiliar neste cálculo, você deve obter $dtheta = \frac{\partial J}{\partial \theta} = x$.

In [35]:
```
# FUNÇÃO DE AVALIAÇÃO: PROPAGAÇÃO PARA TRÁS

def backward_propagation(x, theta):
    """
    Computa a derivada de J com relação a theta (Veja a Figura 1).

    Argumentos:
    x -- uma entrada de valor real
    theta -- nosso parâmetro, também um número real

    Retorna:
    dtheta -- o gradiente do custo com relação a theta
    """

    ### INICIE O SEU CÓDIGO AQUI ### (approx. 1 linha)
    dtheta = x
    ### TÉRMINO DO CÓDIGO ###

    return dtheta
```

In [36]:
```
x, theta = 2, 4
dtheta = backward_propagation(x, theta)
print ("dtheta = " + str(dtheta))
```

```
dtheta = 2
```

Saída esperada:

** dtheta **	2

Exercício: Para mostrar que a função `backward_propagation()` está computando corretamente a derivada $\frac{\partial J}{\partial \theta}$, vamos implementar a verificação de gradiente.

Instruções:

- Primeiro determine "gradapprox" usando a fórmula acima (1) e um valor pequeno para ε. Aqui estão as etapas a serem seguidas:
  1. $\theta^{+} = \theta + \varepsilon$
  2. $\theta^{-} = \theta - \varepsilon$
  3. $J^{+} = J(\theta^{+})$
  4. $J^{-} = J(\theta^{-})$
  5. $gradapprox = \frac{J^{+} - J^{-}}{2\varepsilon}$
- Depois, determine o valor do gradiente usando a função de propagação para trás e armazene o resultado na variável "grad"
- Finalmente, compute a diferença relativa entre "gradapprox" e "grad" usando a seguinte fórmula:

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

  Você irá precisar de 3 etapas para computar esta fórmula:
  - 1'. Determine o numerador utilizando np.linalg.norm(...)
  - 2'. Determine o denominador. Você precisa chamar a função np.linalg.norm(...) duas vezes.
  - 3'. Execute a divisão.
- Se a diferença for pequena (da ordem de 10^{-7}), você pode ficar confidente de que sua implementação da propagação para trás está correta. Caso contrário, pode existir algum erro na computação da derivada.

In [37]:
```
# FUNÇÃO DE AVALIAÇÃO: verificação de gradiente

def gradient_check(x, theta, epsilon = 1e-7):
    """
    Implemente a propagação para trás apresentada na Figura 1.

    Argumentos:
    x -- uma entrada de valor real
    theta -- nosso parâmetro, também um valor real
    epsilon -- pequeno valor utilizado para aproximar o gradiente com a
fórmula (1)

    Retorna:
    difference -- diferença entre (2) o gradiente aproximado e o gradien
te determinado na propagação para trás.
    """

    # Compute gradapprox using o lado esquerdo da formula (1). epsilon é
pequeno o bastante, você não precisa se preocupar
    # sobre o limite.
    ### INÍCIO DO CÓDIGO AQUI ### (approx. 5 linhas)
    theta_plus = theta + epsilon                                # Step 1
    theta_minus = theta - epsilon                               # Step 2
    J_plus = forward_propagation(x, theta_plus)
# Step 3
    J_minus = forward_propagation(x, theta_minus)
# Step 4
    gradapprox = (J_plus - J_minus) / (2 * epsilon)
# Step 5
    ### TÉRMINO DO CÓDIGO ###

    # Verifique se gradapprox é próximo o bastante da saída da propagaçã
o para trás
    ### INÍCIO DO CÓDIGO AQUI ###  (approx. 1 linha)
    grad = backward_propagation(x, theta)
    ### TÉRMINO DO CÓDIGO ###

    ### INÍCIO DO CÓDIGO AQUI ###  (approx. 3 linhas)
    numerador = np.linalg.norm(grad -  gradapprox)
# Step 1'
    denominador = np.linalg.norm(grad) + np.linalg.norm(gradapprox)
# Step 2'
    difference = numerador / denominador                              # S
tep 3'
    ### TÉRMINO DO CÓDIGO ###

    if difference < 1e-7:
        print ("O gradiente está correto!")
    else:
        print ("O gradiente está errado!")

    return difference
```

In [38]:
```
x, theta = 2, 4
difference = gradient_check(x, theta)
print("diferença = " + str(difference))
```

```
O gradiente está correto!
diferença = 2.919335883291695e-10
```

Expected Output: O gradiente está correto!

** diferença **	2.9193358103083e-10

Parabéns, a diferença é menor que o limite de 10^{-7}. Você tem motivos para estar confidente em sua implementação da propagação para trás.

Agora, num caso mais geral, a sua função de custo J possui mais que um entrada em 1D. Quando você está treinando uma rede neural, θ é um vetor formado pro diversas matrizes $W^{[l]}$ e valores de bias $b^{[l]}$! É importante saber implementar a verificação de gradiente para entradas de grnades dimensões. Vamos implementar este caso!

3) Verificação do gradiente para funções N-dimensionais

A figura abaixo descreve a propagação para frente e para trás do seu modelo de detecção de fraudes.

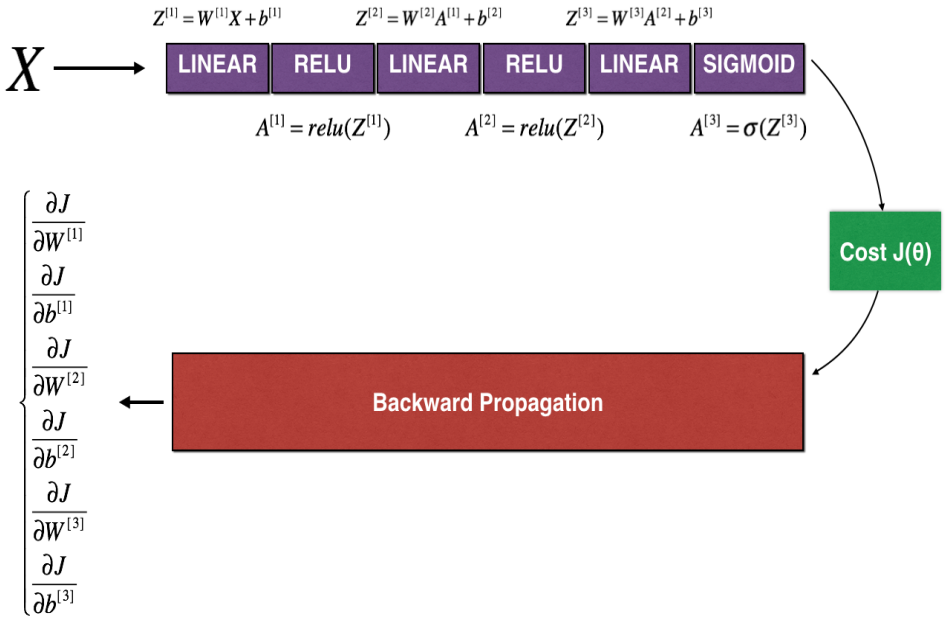

Figura 2: **rede neural profunda**
LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID

Vamos verificar a implementação das propagações para frente e para trás.

In [39]:
```
def forward_propagation_n(X, Y, parameters):
    """
    Implementa a propagação para frente (e determina o valor do custo) a
presentado na Figure 3.

    Argumentos:
    X -- conjunto de treinamento com m exemplos
    Y -- valores de saída para os m exemplos
    parâmetros -- dicionário em python contendo os parâmetros "W1", "b1"
, "W2", "b2", "W3", "b3":
                    W1 -- matriz de pesos no formato (5, 4)
                    b1 -- vetor de bias no formato (5, 1)
                    W2 -- matriz de pesos no formato (3, 5)
                    b2 -- vetor de bias no formato (3, 1)
                    W3 -- matriz de pesos no formato (1, 3)
                    b3 -- vetor de bias no formato (1, 1)

    Retorna:
    cost -- a função de custo (custo logístico para um exemplo)
    """

    # recupere os parâmetros
    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)
    Z3 = np.dot(W3, A2) + b3
    A3 = sigmoid(Z3)

    # Custo
    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
```

Agora, execute a propagação para trás.

In [41]:
```
def backward_propagation_n(X, Y, cache):
    """
    Implementa a propagação para trás apresentada na figure 2.

    Argumentos:
    X -- entrada, no formato (tamanho_da_entrada, 1)
    Y -- saída correta
    cache -- cache das saídas obtidas na propagação para frente

    Retorna:
    gradientes -- Um dicionário com os gradientesdo custo com relação a
cada parâmetro, variáveis de ativação e pre-ativação.
    """

    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
```

Você conseguiu alguns resultados de detecção de fraude do seu conjunto de teste, mas você não está totalmente convencido sobre o seu modelo. Ninguém é perfeito! Vamos implementar a verificação de gradiente para conferir se os gradientes determinados estão corretos.

Como a verificação de gradiente funciona?.

Da mesma forma que em 1) e 2), você quer comparar "gradapprox" com o valor do gradiente computado pela propagação para trás. A fórmula ainda é:

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

Porém, θ não é mais um escalar. É um dicionário chamado de " parâmetros". Nós implementamos uma função "`dictionary_to_vector()`" para você you. Ela converte os "parameters" do dicionário em um vetor chamado de "values", obtido pela reformatação de todos os parâmetros (W1, b1, W2, b2, W3, b3) em vetores e concatenando os vetores.

A função inversa é "`vector_to_dictionary`" que retorna o dicionário de parâmetros.

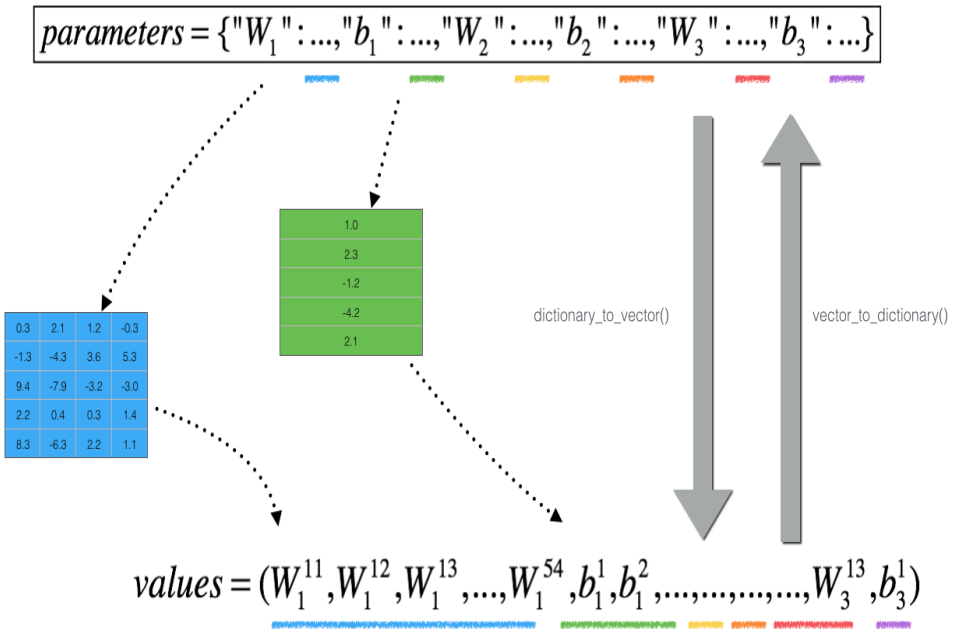

Figura 3: **dictionary_to_vector() e vector_to_dictionary()**
Você irá utilizar estas funções para fazer a verificação do gradiente

Nós ainda convertemos o dicionário de gradientes em um vetor "grad" usando gradients_to_vector(). Você não precisa se preocupar com isto.

Exercício: Implemente gradient_check_n().

Instruções: Aqui está o pseudo-código que irá ajudá-lo a implementar a verificação do gradiente.

Para cada i em num_parameters:

- Para determinar `J_plus[i]`:
    1. Faça θ^+ to `np.copy(parameters_values)`
    2. Faça θ_i^+ to $\theta_i^+ + \varepsilon$
    3. Calcule J_i^+ usando a `forward_propagation_n(x, y, vector_to_dictionary(`θ^+ `))`.
- Para determinar `J_minus[i]`: faça a mesma coisa usando θ^-
- Determine $gradapprox[i] = \frac{J_i^+ - J_i^-}{2\varepsilon}$

Você determinou o vetor gradapprox, onde gradapprox[i] é uma aproximação do gradiente com relação ao parâmetro na posição "i". Agora é possível comparar o vetor gradapprox com os valores dos gradientes dados pela função de propagação para trás. Da mesma forma que fizemos no caso em 1D (Etapas 1', 2', 3'), compute:
[Math Processing Error]

In [82]:
```
# FUNÇÃO DE AVALIAÇÃO: gradient_check_n

def gradient_check_n(parameters, gradients, X, Y, epsilon = 1e-7):
    """
    Verifica se a propagação para trás computa corretamente o gradiente da função de custo dada pela propagação para frente.

    Argumentos:
    parameters -- dicionário python que contém os parâmetros "W1", "b1", "W2", "b2", "W3", "b3":
    grad -- saída da função backward_propagation_n, contém os gradientes do custo com relação aos parâmetros.
    x -- entrada, no formato (tamanho_da_entrada, 1)
    y -- saída correta
    epsilon -- valor pequeno para determinar o gradiente aproximado pela fórmula(1)

    Retorna:
    difference -- diferença (2) entre o gradiente aproximado e o gradiente determinado pela propagação para trás.
    """

    # Organizando as variáveis
    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))

    # Computando gradapprox
    for i in range(num_parameters):

        # Compute J_plus[i]. Entradas: "parameters_values, epsilon". Output = "J_plus[i]".
        # "_" é utilizado porque a função para o cálculo de J possui duas saídas mas nós estamos interessados
        # apenas na primeira
        ### INICIE O SEU CÓDIGO AQUI ### (approx. 3 linhas)
        theta_plus = np.copy(parameters_values)
# Step 1
        theta_plus[i][0] = theta_plus[i][0] + epsilon
# Step 2
        J_plus[i], _= forward_propagation_n(X, Y, vector_to_dictionary(theta_plus))                                  # Step 3
        ### TÉRMINO D CÓDIGO ###

        # Compute J_minus[i]. Entrada: "parameters_values, epsilon". Saída = "J_minus[i]".
        ### INICIE O SEU CÓDIGO AQUI ### (approx. 3 linhas)
        theta_minus = np.copy(parameters_values)
# Step 1
        theta_minus[i][0] = theta_minus[i][0] -  epsilon
# Step 2
        J_minus[i], _ = forward_propagation_n(X, Y, vector_to_dictionary(theta_minus))                                  # Step 3
        ###  TÉRMINO D CÓDIGO ###

        # Compute gradapprox[i]
        ### INICIE O SEU CÓDIGO AQUI ### (approx. 1 linha)
        gradapprox[i] = (J_plus[i] - J_minus[i]) / (2 * epsilon)
        ###  TÉRMINO D CÓDIGO ###

    # Compare gradapprox com a propagação para trás determinando a diferença.
    ### INICIE O SEU CÓDIGO AQUI ### (approx. 1 linha)

    numerador = np.linalg.norm(grad - gradapprox)
```

In [83]:
```
X, Y, parameters = gradient_check_n_test_case()

cost, cache = forward_propagation_n(X, Y, parameters)
gradients = backward_propagation_n(X, Y, cache)
difference = gradient_check_n(parameters, gradients, X, Y)
```

```
Existe um problema com a diferença na propagação para trás = 0.2850931566
540251
```

Saída esperado:

** Existe um problema com a diferença na propagação para trás = **	0.285093156781

Parece que existe um problema com a função `backward_propagation_n` fornecida! Ainda bem que você fez a verificação de gradiente. Volte para a função `backward_propagation` e tente encontrar/corrigir os erros *(Dica: verifique dW2 e db1)*. Execute novamente a verificação de gradiente quando você acar que resolveu os problemas. Lembre-se de executar novamente a célula que define `backward_propagation_n()` se você modificar o código.

Você consegue fazer as correções necessárias no código? Mesmo não sendo avaliado sugere-se que você procure arrumar o código e rodar novamente o verificador de gradientes.

Nota

- Verificação de gradiente é um processo lento! Aproximar o gradiente com $\frac{\partial J}{\partial \theta} \approx \frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon}$ é caro computacionalmente. Por esta razão nós não executamos esta verificação em cada interação durante o treinamento. Apenas algumas vezes para ter certeza se a propagação para trás está correta.
- A verificação de gradiente, como apresentado, não funciona com dropout. Você deve executar a verificação de gradiente sem o dropout para verificar a propagação para trás e depois adicionar o dropout.

Parabéns, você pode ter certeza de que o seu modelo para detectar fraudes está funcionando corretamente.

O que você deve lembrar deste notebook:
- Verificação de gradiente determina o quanto os gradientes da propagação para trás estão próximos daaproximação numérica dos gradientes.
- Verificação de gradiente é lenta, portanto, não deve ser utilizada em todas asa interações durante o treinamento. Utilize apenas em algumas interações para verificar os valores e, em seguida, deslique o uso da verificação.