

CURSO 1

Neural network

Uma rede neural nada mais é que um conjunto de funções que se alimentam entre si. A partir de determinados inputs, cada neurônio retornará um resultado que pode ou não estar alimentando outro neurônio.

Binary Classification

A **regressão logística** (algoritmo) é usado para classificação. A **classificação binária**, como diz o nome, é usada para classificar se é (1) ou não é (0).

As imagens são armazenadas como 3 matrizes, cada uma representando a intensidade de vermelho, azul e verde em cada pixel.

O que iremos realizar é transformar cada uma destas matrizes em um vetor com cada um dos elementos. Primeiro transformaremos o vetor de vermelho, com cada elemento em uma linha (transformando linha por linha do vetor. Exemplo: [0,0], [0,1], [0,2], [1,0], [1,1]...) até terminar com as 3 cores.

Notações

(x,y) – Um exemplo de treino, onde o **x** pertence aos números reais e pertence ao elemento **n** do vetor de elementos e o **y** é uma classificação (0 ou 1);

m – quantidade de exemplos do teste

X – Matriz de exemplos de testes. Não confundir **matrix** com **vetor**. Dentro deste vetor haverão **m** colunas, sendo **m** a quantidade de exemplos de treino e a quantidade de linhas será **Nx**.

Y – O resultado será em um vetor $1 \times n$, sendo cada coluna a resposta de cada coluna de **X**.

.shape – método do objeto da matriz que retorna o seu tamanho (comando de Python)

Notation

$(x, y) \quad x \in \mathbb{R}^{n_x}, y \in \{0, 1\}$
 m training examples: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$
 $M = M_{\text{train}} \quad M_{\text{test}} = \# \text{test examples.}$

$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}$
 $X \in \mathbb{R}^{n_x \times m}$
 $X.\text{shape} = (n_x, m)$
 $Y = [y^{(1)} \ y^{(2)} \ \dots \ y^{(m)}]$
 $Y \in \mathbb{R}^{1 \times m}$
 $Y.\text{shape} = (1, m)$

Figura 1 - Slide do vídeo "Binary Classification" da primeira semana

Regressão Logística

Dado x , sendo x , neste exemplo, uma imagem, queremos um algoritmo que $\hat{y} = P(y = 1 | x)$. A fórmula quer dizer que: **um algoritmo que presuma se x é ou não o que queremos identificar.** O algoritmo retorna uma probabilidade ($0 \leq y \leq 1$), dado input x , seja o que queremos identificar ou não.

Continuando o exemplo, dado x uma image, queremos saber se x tem a chance de ser uma imagem de um gato ou não.

Dado que x é um vetor de tamanho n (que pertence aos números reais) e os parâmetros (w) também é um vetor de tamanho n (que também pertence aos números reais) e b , que é um número real (que pode ser comparado ao parâmetro **theta0** da regressão linear).

Se tratarmos a saída como $\hat{y} = wTx + b$, sendo T de **transposto** (vetor transposto = transformar linhas em colunas e colunas em linhas), teremos uma função de **regressão linear**, o que não é o intuito do algoritmo, pois o \hat{y} tem que estar entre 0 e 1.

Para que tenhamos o resultado acima citado, temos que $z = \sigma(\hat{y} = wTx + b)$ (função sigmoide)

A **função sigmoide** é definida como $\sigma(z) = \frac{1}{1+e^{-z}}$

Sobre a fórmula da função sigmoide:

- Se z for muito grande ou se o valor for próximo a 0, então teremos $1/1 + 0$, que terá resultado 1;
- Se z for muito pequeno ou for um grande número negativo, o 1 será dividido por um número muito grande, o que acarreta que o resultado será próximo a 0;

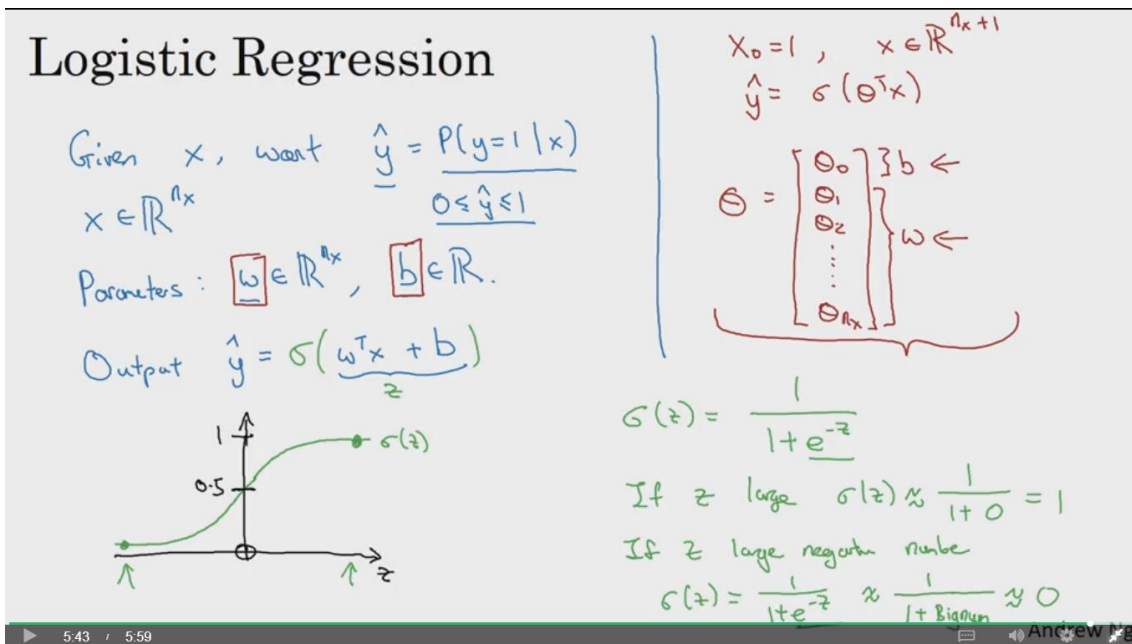


Figura 2- Slide do vídeo "Logistic Regression" da semana 2

Logistic Regression Cost Function

$$\hat{y} = \sigma(w^T x + b)$$

Onde $\sigma(z) = \frac{1}{1+e^{-z}}$

Dado $\{(x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)})\}$, queremos $\hat{y}^{(i)} \approx y^{(i)}$

Sendo:

- A primeira fórmula é a **função sigmoide** para se obter o \hat{y} (a predição de se é 0 ou 1);
- A função sigmoide com parâmetro z
- A parte abaixo é sobre os valores de treino que dado um determinado x , há um y .
- O z nada mais é que a saída da função \hat{y} no começo do vídeo

A **função de perda (loss function)** é usada para medir o quão bem o nosso algoritmo está performando.

Estes tipos de funções são usados para verificar em qual ponto está o **local ótimo** da nossa função. Os bons algoritmos possuem resultados de forma convexa, ou seja, com uma forma de **U**, onde a parte mais baixa do U é a melhor parte.

Para performar a função de regressão logística poderíamos usar a função $L(\hat{y}, y) = \frac{1}{2} (\hat{y} - y)^2$, mas esta função não é muito utilizada, pois pode ter um resultado não-convexo. Estes cálculos são usados para o **gradient descent (gradiente descendente)** para conseguir achar os valores ótimos da função.

A função de perda usada para regressão logística é $L(\hat{y}, y) = -[(y \log \hat{y}) + (1 - y) \log (1 - \hat{y})]$.

O raciocínio aplicado nesta função é:

- Se $y = 0$, então a primeira parte do algoritmo será descartada, pois sendo $y \log \hat{y}$ e $y = 1$, então $0 \log \hat{y} = 0$. Graças a isso queremos que \hat{y} seja grande. Por exemplo, digamos que tenhamos $L(0.5, 0)$, então teremos:
 - $L(0.5, 0) = -[(0 \log 0.5) - (1 - 0) \log (1 - 0.5)]$
 - $L(0.5, 0) = - (1 \log 0.5)$
 - $L(0.5, 0) = -(1 \cdot -0.639)$
 - $L(0.5, 0) = 0.639$
- Se $y = 1$, então a segunda parte do algoritmo será descartada, pois sendo $(1 - y) = 0$, então $\log (1 - \hat{y})$ também será 0. Exemplo:
 - $L(0.5, 1) = -[(1 \log 0.5) - (1 - 1) \log (1 - 0.5)]$
 - $L(0.5, 1) = - (1 \cdot -0.639)$
 - $L(0.5, 1) = 0.639$

O que podemos observar é que o resultado será “o mesmo”, sendo y 0 ou 1. Lembrando que y é a classificação original. Isso quer dizer que, se estamos falando de “imagens de gatos”, 1 é imagem de gato e 0 é algo que **NÃO** é uma imagem de gato.

Já a **função de custo** é uma média da soma da **função de perda**. Em fórmula podemos escrever que $J(w, b) = -\frac{1}{m} \sum_{i=1}^m L(y^{(i)}, \hat{y}^{(i)})$, sendo L igual a **função de custo** escrita acima.

Logistic Regression cost function

$\rightarrow \hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$, where $\sigma(z^{(i)}) = \frac{1}{1 + e^{-z^{(i)}}}$ $z^{(i)} = w^T x^{(i)} + b$

Given $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$. $\begin{matrix} x^{(i)} \\ y^{(i)} \\ z^{(i)} \end{matrix}$ i -th example.

Loss (error) function: $\mathcal{L}(\hat{y}, y) = \frac{1}{2} (\hat{y} - y)^2$

$\mathcal{L}(\hat{y}, y) = - (y \log \hat{y} + (1 - y) \log (1 - \hat{y}))$

If $y = 1$: $\mathcal{L}(\hat{y}, y) = - \log \hat{y} \leftarrow$ Want $\log \hat{y}$ large, want \hat{y} large.

If $y = 0$: $\mathcal{L}(\hat{y}, y) = - \log (1 - \hat{y}) \leftarrow$ Want $\log (1 - \hat{y})$ large ... want \hat{y} small

Cost function: $J(w, b) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log (1 - \hat{y}^{(i)})]$

Figura 3- Slide da aula "Logistic Regression Cost Function" da primeira semana

Gradient descent

O **gradient descent** é usado para minimizar o custo de $J(w, b)$. Conforme o algoritmo vai rodando, ele vai atualizando os parâmetros w e b até que eles sejam os menores possíveis. Menores no sentido que eles se ajustem ao **custo ótimo** da função.

Ao visualizar os possíveis resultados como um gráfico convexo, podemos raciocinar o **gradient descent** partindo de um ponto do gráfico até a parte mais baixa, onde é o **global optimal**.

Ao iniciar o algoritmo, tanto faz com qual valor os parâmetros são inicializados, pois o objetivo do algoritmo é sempre chegar a posição ótima. Pelo que o professor Ng disse, a maioria das pessoas que usam esse algoritmo iniciam os parâmetros com valor **0**.

O cálculo do gradient descent é: $w := w - \alpha \frac{dJ(w)}{dw}$

Este cálculo é repetido **N** vezes até que o algoritmo chegue ao seu valor ótimo. No exemplo acima não está sendo exibido o **b**. O alfa acima é o **fator de aprendizado**, que nada mais é que o quanto o algoritmo vai se deslocando até chegar ao valor ótimo.

O cálculo acima é uma **derivada de um parâmetro**. O que isso tem a ver? Vejamos como vamos escrever isso usando dois parâmetros:

$$w := w - \alpha \frac{\partial J(w, b)}{\partial w}; b := b - \alpha \frac{\partial J(w, b)}{\partial b}$$

Vejam que a letra inicial mudou, mas o cálculo continua o mesmo. A diferença é que, em notação matemática, o símbolo de uma derivada **com um parâmetro é um simples "d"**; quando a derivada possui **mais de um parâmetro, ela possui esse ∂** , que nada mais é que um "d" com outra fonte.

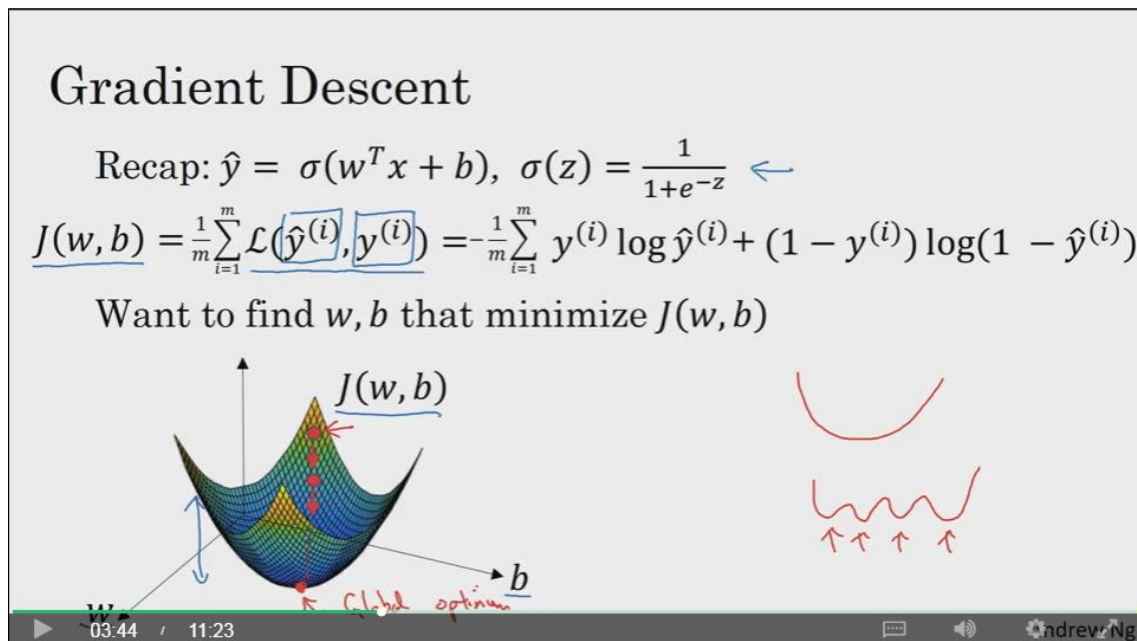


Figura 4 - Slide 1 do vídeo "Gradient Descent" da semana 2

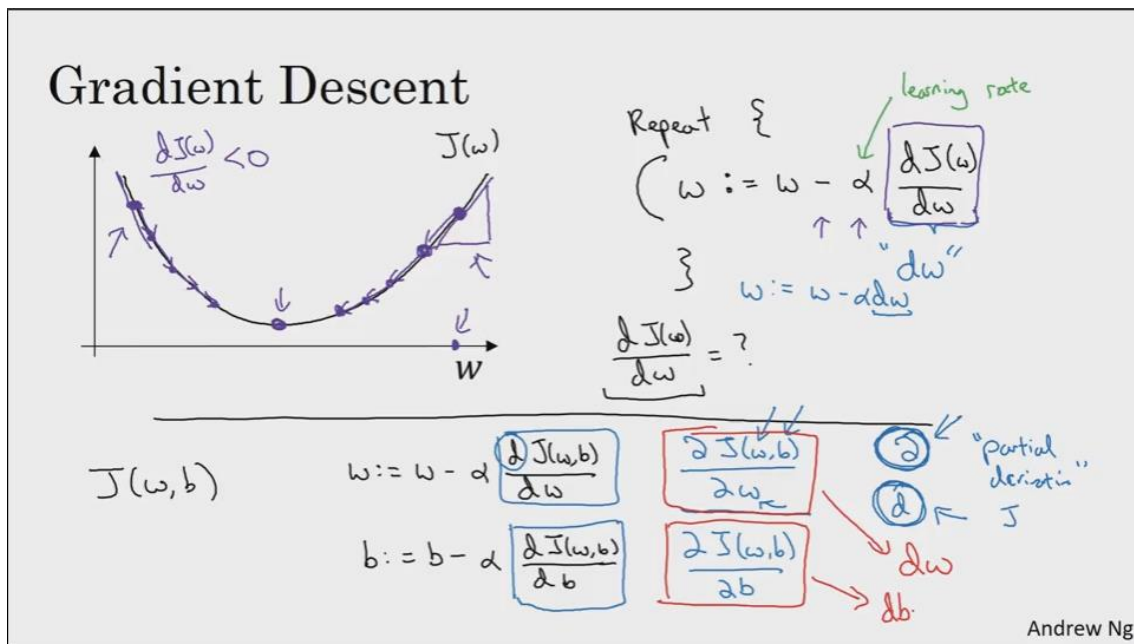


Figura 5 - Slide 2 do vídeo "Gradient Descent" da semana 2

Derivadas

As **derivadas** nada mais são do que a diferença entre o **x** e **y** de um ponto e o **x** e **y** de outro ponto. Exemplo:

Caso tenhamos uma função **$f(1) = 3$** , que é **linear**, e queiramos obter a derivada, que é a diferença entre o ponto atual e um após (ou antes, o importante é a diferença entre os pontos), verificamos, literalmente falando, a diferença entre um ponto após. Vemos que **$f(2) = 6$** . Se passarmos isso para um gráfico...

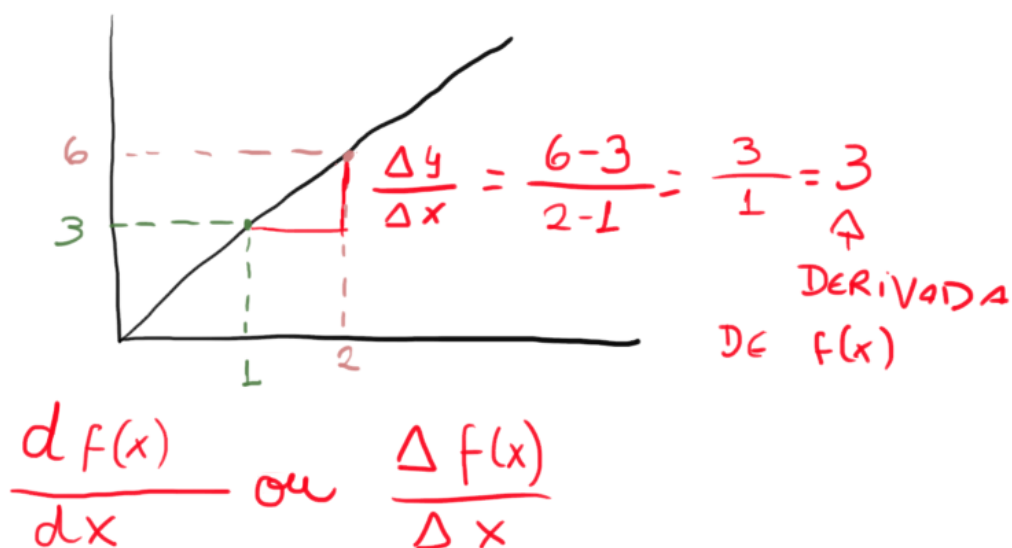


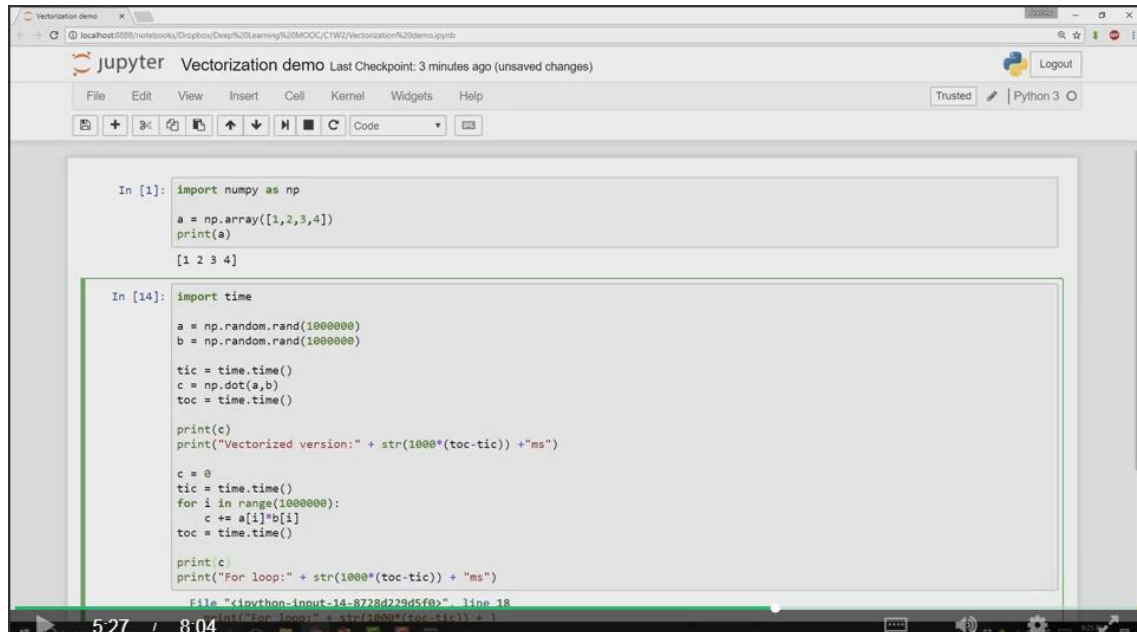
Figura 6 - Representação do cálculo de uma derivada em uma função linear

Isto quer dizer que qualquer mudança em **x** será multiplicada por 3 em **y**.

Vetorização

Utilizar a multiplicação de vetores é muito mais rápido do que usar um laço **for** para realizar a ação. Pelo exemplo que o professor Ng deu no vídeo, a velocidade do cálculo vetorizado chega a ser mais de 300x mais rápida do que com um laço **for**.

Usando da biblioteca **numpy** é possível obter tal multiplicação com o auxílio da função **np.dot**.



```
In [1]: import numpy as np
a = np.array([1,2,3,4])
print(a)
[1 2 3 4]

In [14]: import time
a = np.random.rand(1000000)
b = np.random.rand(1000000)
tic = time.time()
c = np.dot(a,b)
toc = time.time()

print(c)
print("Vectorized version:" + str(1000*(toc-tic)) + "ms")

c = 0
tic = time.time()
for i in range(1000000):
    c += a[i]*b[i]
toc = time.time()

print(c)
print("For loop:" + str(1000*(toc-tic)) + "ms")
```

Figura 7 - Print do vídeo "Vectorization" da semana 2

Um pouco mais sobre vetorização

É possível também pensar na vetorização na parte dos cálculos dos parâmetros para calcular o **gradient descent**.

Vetorizando a regressão logística

Ao contrário de ter que implementar a **regressão logística** em vários lops, é possível vetorizar os parâmetros da função e executá-los de uma só vez.

Caso tenhamos um vetor com tamanho **m** e **n** colunas, ao invés de ficar iterando de linha e coluna, o que acarretaria em dois **for**, podemos transpor a matriz para que os elementos de cada treino estejam no mesmo vetor.

A matriz de treino é composta pelos elementos da cada um, sendo cada coluna um atributo e cada linha uma observação do dataset de treino. Ao transformarmos cada coluna em linha, conseguimos multiplicar todos os elementos através de uma multiplicação de vetores.

No código podemos colocar cada um dos resultados como a letra maiúscula de cada um dos elementos calculados. Se calculamos os elementos do vetor \mathbf{x} , então o resultado desta transformação será nomeada \mathbf{X} e assim por diante.

Vectorizing Logistic Regression

$\rightarrow \underline{z^{(1)}} = \underline{w^T x^{(1)}} + b$
 $\rightarrow \underline{a^{(1)}} = \sigma(z^{(1)})$

$\underline{z^{(2)}} = \underline{w^T x^{(2)}} + b$
 $\underline{a^{(2)}} = \sigma(z^{(2)})$

$\underline{z^{(3)}} = \underline{w^T x^{(3)}} + b$
 $\underline{a^{(3)}} = \sigma(z^{(3)})$

$\underline{X} = \begin{bmatrix} x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \\ | & | & & | \\ | & | & & | \end{bmatrix} \quad \begin{matrix} (n_x, m) \\ \mathbb{R}^{n_x \times m} \end{matrix}$

$\underline{w} = \begin{bmatrix} w_0 \\ | \\ | \\ | \end{bmatrix} \quad \begin{matrix} (1, n_x) \\ \mathbb{R}^{1 \times n_x} \end{matrix}$

$\underline{z} = \begin{bmatrix} z^{(1)} & z^{(2)} & \dots & z^{(m)} \end{bmatrix} = \underline{w^T X} + \underline{[b \ b \dots b]}$

$\rightarrow \underline{z} = \text{np.dot}(w.T, X) + b$

$\underline{A} = \begin{bmatrix} a^{(1)} & a^{(2)} & \dots & a^{(m)} \end{bmatrix} = \sigma(\underline{z})$

"Broadcasting"

Andrew Ng

Figura 8 - Slide do vídeo "Vectorizing Logistic Regression" da semana 2

Dicas sobre como usar o numpy

Dependendo de como for criado o vetor/matriz, a variável pode se tornar um **"rank 1 array"**. O problema desta variável é que ela tem comportamentos estranhos dependendo do método aplicado nela.

Para evitar este tipo de erro, sempre verificar o **shape** (formato) da variável ao qual você irá trabalhar e, se puder, sempre definir a quantidade de linhas e colunas que seu vetor terá.

Python/numpy vectors

```
a = np.random.randn(5)
a.shape = (5,)
"rank 1 array" } Don't use
```



```
a = np.random.randn(5,1) → a.shape = (5,1) column vector ✓
a = np.random.randn(1,5) → a.shape = (1,5) row vector ✓
```



```
assert(a.shape == (5,1)) ←
a = a.reshape((5,1))
```

Andrew Ng

Figura 9 - Slide do vídeo "A note on python/numpy vectors" da semana 2

Neural Networks Overview

Passos da rede neural:

- Receber os inputs das variáveis, no caso dos exemplos, alguns vetores já tratados e normalizados;
- Obter o resultado da função sigmoide ($\sigma = \frac{1}{1+e^{-z}}$), sendo $z = w^T X + b$
- Obter a função de perda (**loss function**), que é $L(\hat{y}, y) = - [(y \log \hat{y}) + (1 - y) \log (1 - \hat{y})]$
- Calcular o gradient descent dos parâmetros **w** e **b**;
- Se o resultado for maior que 0,5 = 1; 0

Usar $X^{[1]}$ para denotar qual a camada da rede neural.

! Não confundir com $x^{(i)}$, que é usado para denotar valores individuais de treino.

Quando houver mais camadas (exemplo com uma rede de duas camadas):

- Computar o z da camada 1, sendo **x** os parâmetros passados pelas variáveis;
- Computar a função sigmoide da camada 1;
- O resultado da sigmoide da camada 1 será o **x** da camada 2;
- Computar a função de perda, sendo **a** o resultado do sigmoide da camada 2;
- Calcular o gradient descent das camadas inferiores;

What is a Neural Network?

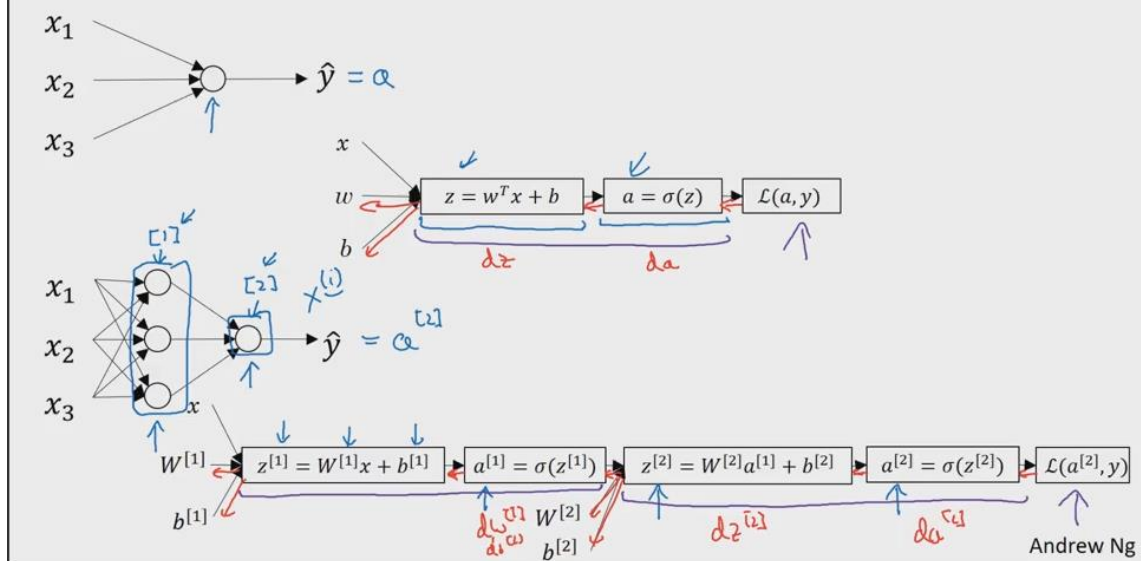


Figura 10 - Slide do vídeo "Neural Network Overview" da semana 3

Neural Network Representation

Em redes neurais, tecnicamente existem três camadas: a camada de entrada (as variáveis que entram na função), as camadas ocultas (as que processam tal informação) e as camadas de saída.

Quando contamos as camadas da rede neural, não contamos a camada de entrada. Ou seja, no exemplo abaixo temos uma rede neural com duas camadas.

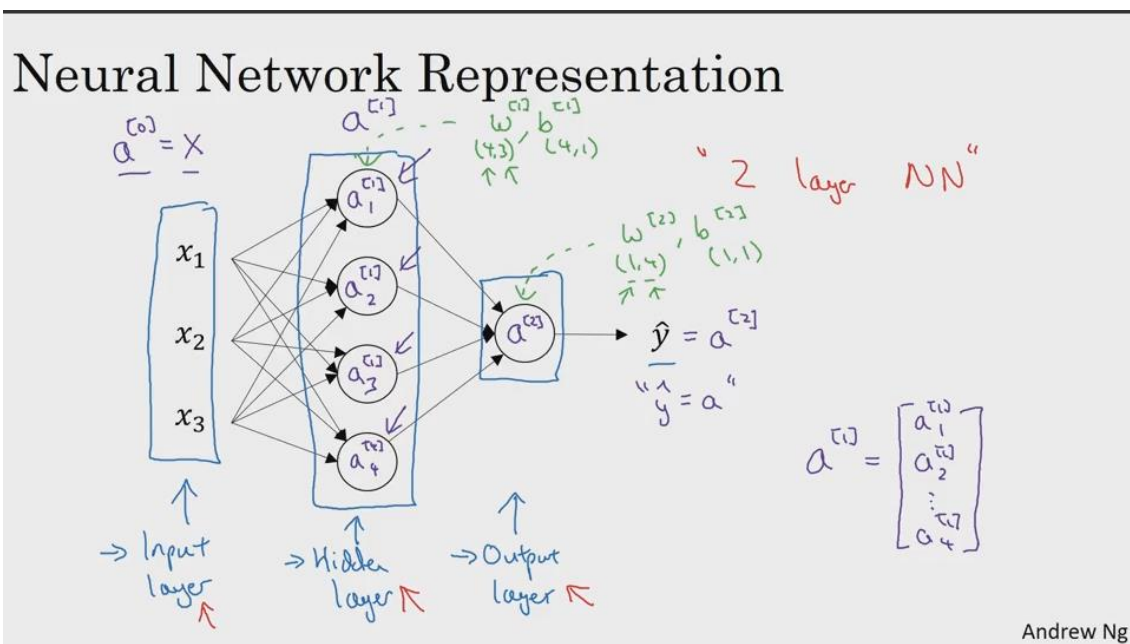


Figura 11 - Slide do vídeo "Neural Network Representation" da semana 3

Computing a Neural Network's Output

Quando tratamos de um algoritmo de classificação, tal qual a regressão logística, cada neurônio será constituído de dois cálculos: computar o **z** e computar o **a**.

O **z** é o cálculo $\mathbf{w}^T \mathbf{x} + \mathbf{b}$, o **a** é o resultado da função sigmoide, sendo $\sigma(\mathbf{z}) = \frac{1}{1 + e^{-\mathbf{z}}}$ que é a função de ativação. O resultado destes dois cálculos é que determinará a predição \hat{y} (isso se tratando de uma rede com um nó).

Notamos que para cada vetor **x**, ele irá ser multiplicado pelo **wT** e o resultado será somado com **b**, que nada mais é que um valor inteiro.

Na questão da notação, temos $z_i^{[l]}$, sendo **[l]** o número da camada e **i** o nó em cada camada.

Dependendo de quantas camadas forem usadas para a rede neural, processar as informações usando de loops (**for**, **while**) pode ser muito lento, por isso é interessante vetorizar os elementos a serem computados e os de entrada.

Pra que isso seja realizado é preciso que empilhemos os valores de **wT** e os armazenemos em uma matriz. Exemplo: Consideremos os vetores abaixo:

w[1]	w[2]	w[3]
1	2	3
4	5	6
7	8	9

Após transposição e empilhamento, o vetor se tornará a seguinte matriz **W**

1	4	7
2	5	8
3	6	9

Sendo que há 3 entradas em um vetor [1,3]

10
20
30

Daí teremos dois casos: o que multiplicaremos elemento por elemento e o que teremos uma multiplicação de vetores. O resultado do primeiro será um vetor **3x3** ($\mathbf{w.T} * \mathbf{x}$), já o segundo terá um vetor **3x1** ($\text{np.dot}(\mathbf{w.T}, \mathbf{x})$).

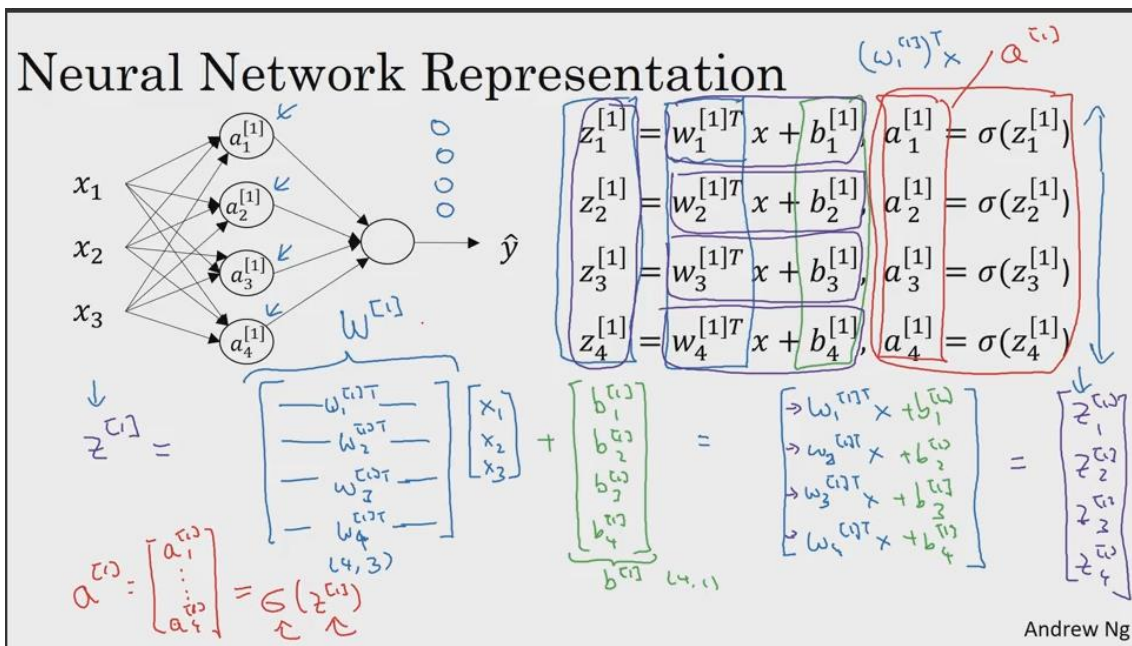


Figura 12 - Slide 1 do vídeo "Computing a Neural Network's Output" da semana 3

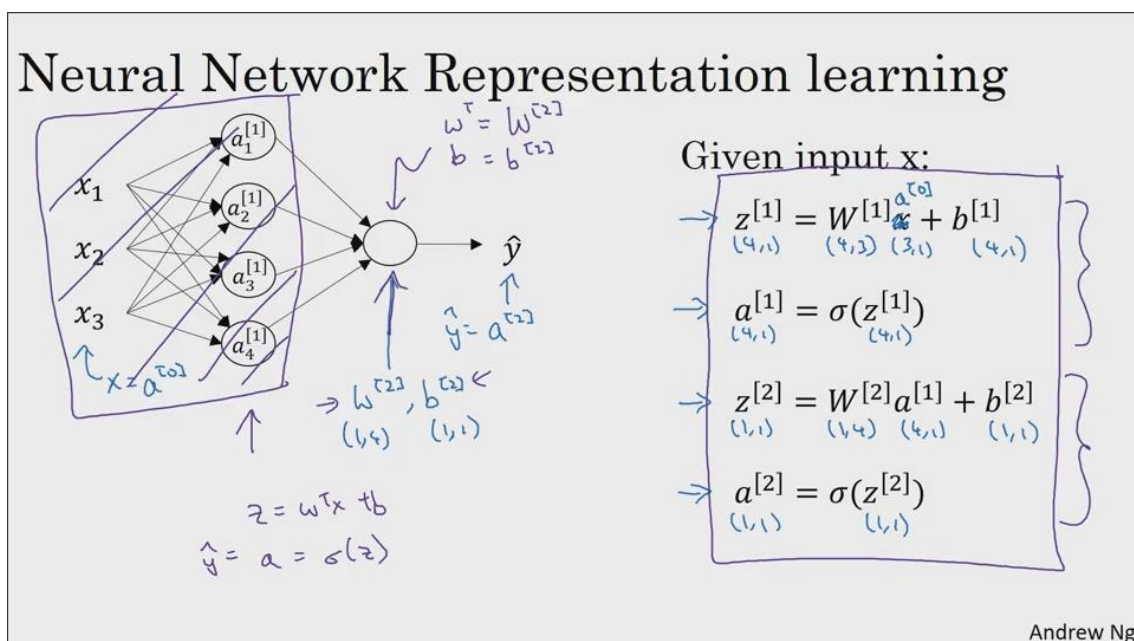


Figura 13- Segundo slide do vídeo "Computing a Neural Network's Output" da semana 3

Vectorizing across multiple examples

Uma das coisas que temos que pensar é que cada nó é um "neurônio" diferente e, por causa disso, possui uma função diferente no nosso cálculo. Isso quer dizer que cada nó terá um parâmetro w e um b , pois cada um é independente do outro. Quando vetorizamos os parâmetros, vetorizamos todas as entradas (x), todos os parâmetros de pesos (w) e todos os valores b .

Ao invés de verificarmos os resultados através de um loop, podemos empilhar os exemplos de treino e utilizar os resultados dentro de uma matriz. Exemplo: após empilhar os exemplos de

treino \mathbf{w} (empilhar de forma transposta), teremos uma matriz \mathbf{W} ; os parâmetros \mathbf{x} serão vetorizados: se forem do tamanho \mathbf{xyz} , se transformarão em um vetor $[\mathbf{xyz}, 1]$; os parâmetros \mathbf{b} também serão dispostos na mesma ordem, criando a matriz \mathbf{B} .

Não podemos esquecer que a quantidade de linhas tem que ser igual a quantidade de colunas para que a multiplicação de matrizes sejam feitas de forma correta.

Pela notação descrita na imagem abaixo, temos que interpretar os vetores da seguinte forma: exemplos do treino (horizontalmente), nós ocultos (verticalmente).

	Ex. 1	Ex. 2	Ex. 3
Nó 1	1	2	3
Nó 2	4	5	6
Nó 3	7	8	9

Exemplo de tabela de um resultado \mathbf{A}

Vectorizing across multiple examples

for $i = 1$ to m :

$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$$

$$a^{[1](i)} = \sigma(z^{[1](i)})$$

$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$$

$$a^{[2](i)} = \sigma(z^{[2](i)})$$

$X = \begin{bmatrix} x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ 1 & 1 & \dots & 1 \end{bmatrix}$

$W^{[1]} = \begin{bmatrix} w^{1} & w^{[1](2)} & \dots & w^{[1](n_x+1)} \end{bmatrix}$

$b^{[1]} = \begin{bmatrix} b^{1} \\ b^{[1](2)} \end{bmatrix}$

$z^{[1]} = \begin{bmatrix} z^{1} & z^{[1](2)} & \dots & z^{[1](m)} \end{bmatrix}$

$A^{[1]} = \begin{bmatrix} a^{1} & a^{[1](2)} & \dots & a^{[1](m)} \\ 1 & 1 & \dots & 1 \end{bmatrix}$

$W^{[2]} = \begin{bmatrix} w^{[2](1)} & w^{2} & \dots & w^{[2](n_h+1)} \end{bmatrix}$

$b^{[2]} = \begin{bmatrix} b^{[2](1)} \\ b^{2} \end{bmatrix}$

$z^{[2]} = \begin{bmatrix} z^{[2](1)} & z^{2} & \dots & z^{[2](m)} \end{bmatrix}$

$A^{[2]} = \begin{bmatrix} a^{[2](1)} & a^{2} & \dots & a^{[2](m)} \\ 1 & 1 & \dots & 1 \end{bmatrix}$

$z^{[1]} = W^{[1]}X + b^{[1]}$
 $\rightarrow A^{[1]} = \sigma(z^{[1]})$
 $\rightarrow z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$
 $\rightarrow A^{[2]} = \sigma(z^{[2]})$

X (input examples) \rightarrow $W^{[1]}$ (weights) \rightarrow $z^{[1]}$ (net input) \rightarrow $A^{[1]}$ (activation) \rightarrow $W^{[2]}$ (weights) \rightarrow $z^{[2]}$ (net input) \rightarrow $A^{[2]}$ (activation)

n_x, m (input size, number of examples)
 n_h (hidden units)

Andrew Ng

Figura 14- Slide do vídeo "Vectorizing across multiple examples" da semana 3

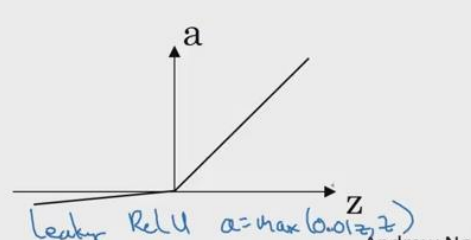
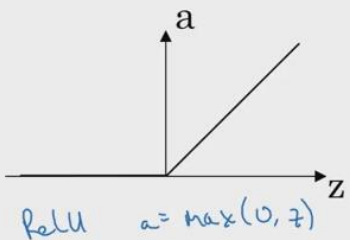
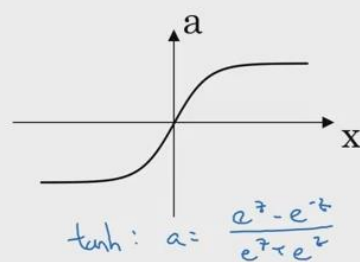
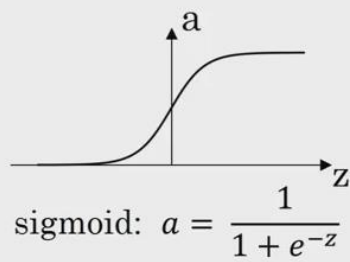
Activation functions

Não existe somente a função sigmoide ($\frac{1}{1 + e^{-z}}$) como função de ativação. Existe a função **tangente hiperbólica** ($\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$), que pode ter valores entre -1 e 1 ou o **ReLU (Rectified linear function – função linear retificada)**, que é $\text{ReLU} = \max(0, z)$.

A tangente hiperbólica é mais eficiente do que a sigmoide, menos quando a unidade final é de classificação binária ("cachorro ou não-cachorro", por exemplo).

O problema da função hiperbólica é quando z é muito grande ou muito pequeno, a derivada da função demora para alterar.

Pros and cons of activation functions



Andrew Ng

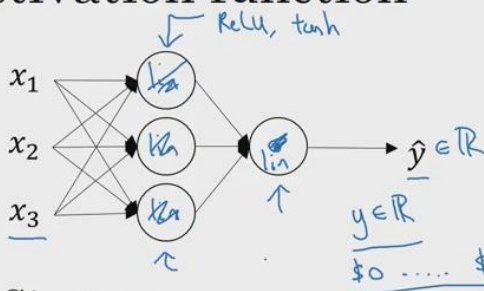
Figura 15 - Slide do vídeo "Activation Function" da semana 3

Why do you need non-linear activation function?

Se não usarmos valores não-lineares entre os nós, então não é necessário acrescentar mais nós, pois dá para simplesmente multiplicarmos as funções lineares.

O único local que poderíamos usar uma função de ativação linear é no nó de saída e em funções de regressão (preço de casas, por exemplo). Ainda assim, além de também usar funções de ativação não-lineares em nós anteriores, como preço de casas não podem ser negativos, poderíamos usar um **ReLU** no lugar.

Activation function



Given x :

- $z^{[1]} = W^{[1]}x + b^{[1]}$
- $a^{[1]} = g^{[1]}(z^{[1]}) = z^{[1]}$
- $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
- $a^{[2]} = g^{[2]}(z^{[2]}) = z^{[2]}$

$g(z) = z$
"linear activation function"

$$\begin{aligned}
 a^{[1]} &= z^{[1]} = W^{[1]}x + b^{[1]} \\
 a^{[2]} &= z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \\
 a^{[2]} &= W^{[2]}(W^{[1]}x + b^{[1]}) + b^{[2]} \\
 &= (W^{[2]}W^{[1]})x + (W^{[2]}b^{[1]} + b^{[2]}) \\
 &= W'x + b' \\
 g(z) &= z
 \end{aligned}$$

Andrew Ng

Figura 16 - Slide do vídeo "Why do you need non-linear activation function?" da semana 3

Sigmoid activation function

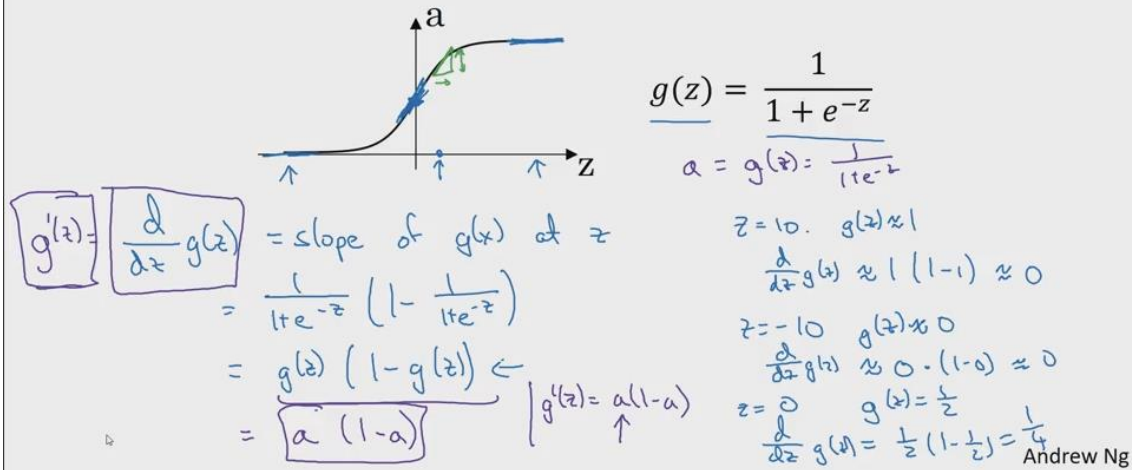


Figura 17- Slide 1 do vídeo "Derivatives of activation function" da semana 3

Tanh activation function

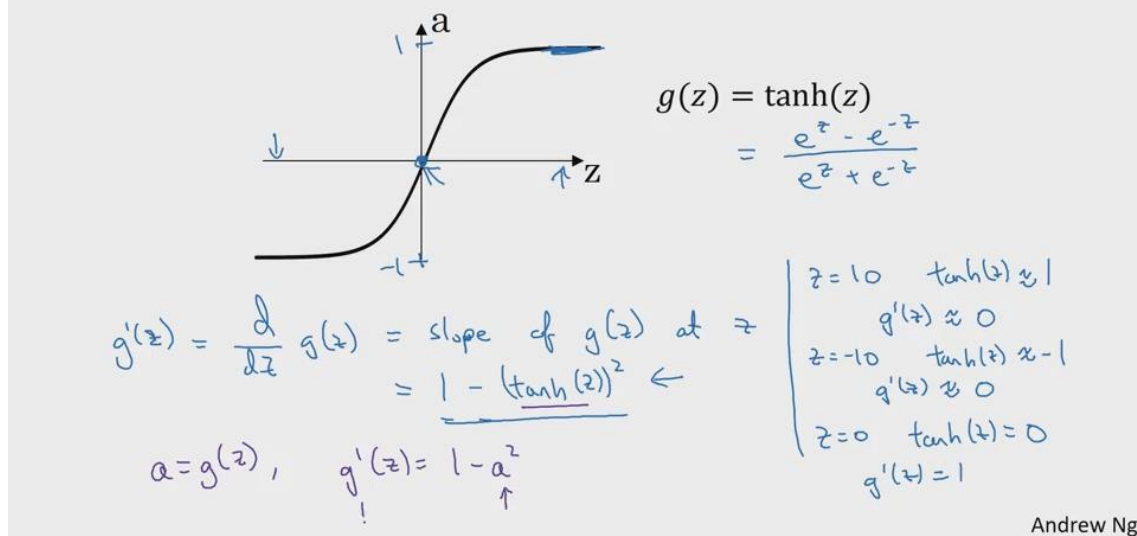


Figura 18 - Slide 2 do vídeo "Derivatives of activation function" da semana 3

$$g(z) = \max(0, z)$$
$$g(z) = \max(0.01z, z)$$

Andrew Ng

Parameters: $\omega^{[1]}$, $b^{[1]}$, $\omega^{[2]}$, $b^{[2]}$
 $(\omega^{[1]}, n^{[1]})$, $(\omega^{[2]}, n^{[2]})$, $(n^{[1]}, 1)$, $(n^{[2]}, 1)$ $n_x = n^{[0]}$, $n^{[1]}$, $\underline{n^{[2]} = 1}$

Cost function: $J(W^{[1]}, b^{[1]}, \underline{W}^{[2]}, \underline{b}^{[2]}) = \frac{1}{n} \sum_{i=1}^n \ell(\hat{y}_i, y_i)$

Gradient Descent:

Report {

Compute $\text{pred}_{\text{net}}(\hat{y}^{(i)}, i=1, \dots, m)$
 $dW^{(1)} = \frac{\partial J}{\partial w^{(1)}}, \quad db^{(1)} = \frac{\partial J}{\partial b^{(1)}}, \dots$

$$W^{(i)} := W^{(i)} - \alpha \partial W^{(i)}$$

$$b^{(1)}_{(2)} = b^{(1)}_{(2)} - \alpha \frac{d}{d} b^{(1)}_{(2)}$$

Figura 20 - Slide 2 do vídeo "Gradient descent for Neural Networks" da semana 3

Formulas for computing derivatives

<p>Forward propagation:</p> $z^{[1]} = w^{[1]}x + b^{[1]}$ $a^{[1]} = g^{[1]}(z^{[1]}) \leftarrow$ $z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$ $a^{[2]} = g^{[2]}(z^{[2]}) = \sigma(z^{[2]})$	<p>Back propagation:</p> $dz^{[2]} = A^{[2]} - Y \leftarrow$ $dw^{[2]} = \frac{1}{m} dz^{[2]} A^{[1]T}$ $db^{[2]} = \frac{1}{m} \text{np.sum}(dz^{[2]}, \text{axis}=1, \text{keepdims}=\text{True})$ $dz^{[1]} = \underbrace{w^{[2]T}}_{(n^{[1]}, m)} dz^{[2]} \times \underbrace{g^{[2]'}(z^{[1]})}_{\text{element-wise product } (n^{[1]}, m)}$ $dw^{[1]} = \frac{1}{m} dz^{[1]} x^T$ $db^{[1]} = \frac{1}{m} \text{np.sum}(dz^{[1]}, \text{axis}=1, \text{keepdims}=\text{True})$	$Y = [y^{(1)} \ y^{(2)} \dots \ y^{(m)}]$ $(n^{[2]}) \leftarrow$ $(n^{[2]}, 1) \leftarrow$
--	--	--

Figura 21 - Slide 2 do vídeo "Gradient descent for Neural Networks" da semana 3

Backpropagation intuition

Summary of gradient descent

$dz^{[2]} = a^{[2]} - y$ $dW^{[2]} = dz^{[2]} a^{[1]T}$ $db^{[2]} = dz^{[2]}$ $dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$ $dW^{[1]} = dz^{[1]} x^T$ $db^{[1]} = dz^{[1]}$	$dZ^{[2]} = A^{[2]} - Y$ $dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$ $db^{[2]} = \frac{1}{m} \text{np.sum}(dZ^{[2]}, \text{axis}=1, \text{keepdims}=\text{True})$ $dZ^{[1]} = \underbrace{W^{[2]T}}_{(n^{[1]}, m)} dZ^{[2]} * \underbrace{g^{[1]'}(Z^{[1]})}_{\text{element-wise product } (n^{[1]}, m)}$ $dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$ $db^{[1]} = \frac{1}{m} \text{np.sum}(dZ^{[1]}, \text{axis}=1, \text{keepdims}=\text{True})$	$J(\cdot) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}_i, y_i)$
--	---	---

Andrew Ng

Figura 22 - Slide do vídeo "Backpropagation intuition" da semana 3

Random initialization

Dependendo do tamanho da sua rede neural, é necessário iniciar os **W** de forma aleatória, pois se forem iniciados com 0, todos os outros nós com os mesmos parâmetros estarão retornando o mesmo resultado. Se for para isso acontecer, não é necessário mais de um nó computacional.

Podemos usar `np.random.randn(x,y) * 0.01` para obter os números aleatórios para **W**. O **b** pode ser inicializado com zeros. Já a constante 0.01 pode ser diferente de acordo com a quantidade de nós que sua aplicação tiver.

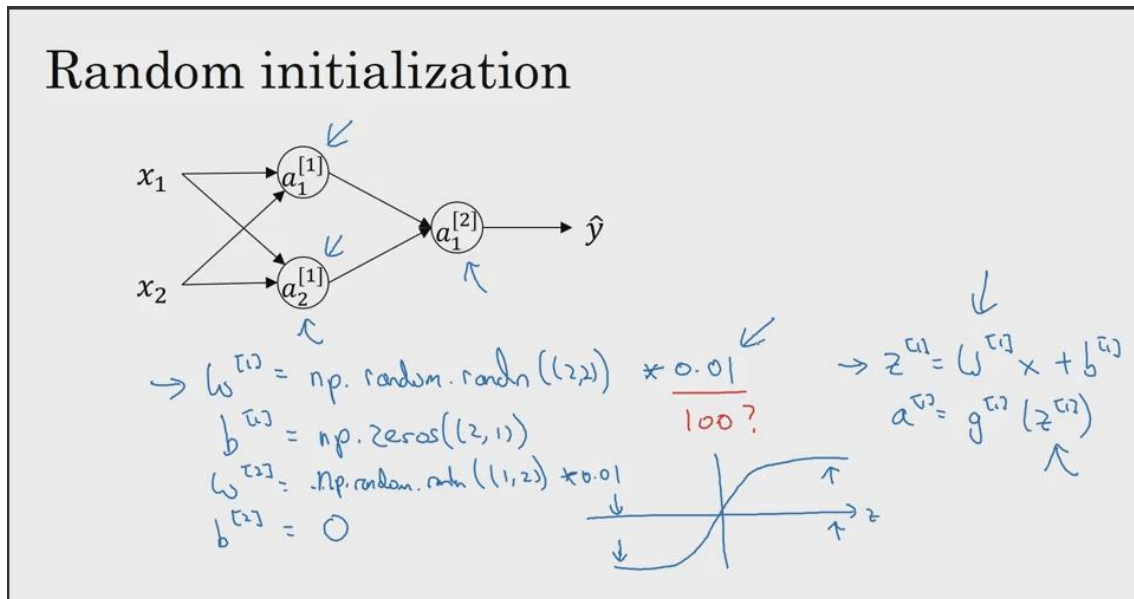


Figura 23 - Slide do vídeo "Random initialization" da semana 3

Deep L-Layer neural network

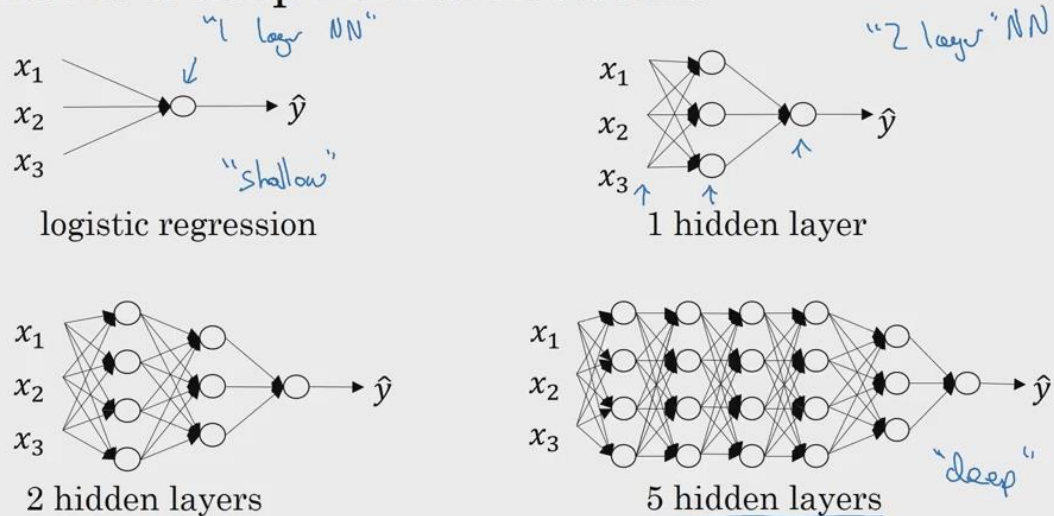
O conceito de deep neural network é definido pela quantidade de camadas de nós (neurônios) que a rede possui. Um nó com uma regressão logística na ponta é uma rede rasa. Conforme a quantidade de camadas vai crescendo, a rede vai se tornando cada vez mais profunda.

A questão da profundidade da rede é que, dependendo de quão rasa a rede for, ela não conseguiria "aprender" determinadas funções. A quantidade de camadas para ter uma boa rede é mais de tentativa e erro.

Pelo fato da rede nem sempre ter a quantidade de nós iguais em cada camada, temos que usar uma **notação** para distinguir cada um deles. As notações são:

- **L** – Quantidade de camadas que temos na rede neural.
- **$n^{[l]}$** – Quantidade de nós na camada **l**;
- **$a^{[l]}$** – Ativação na camada **l**;

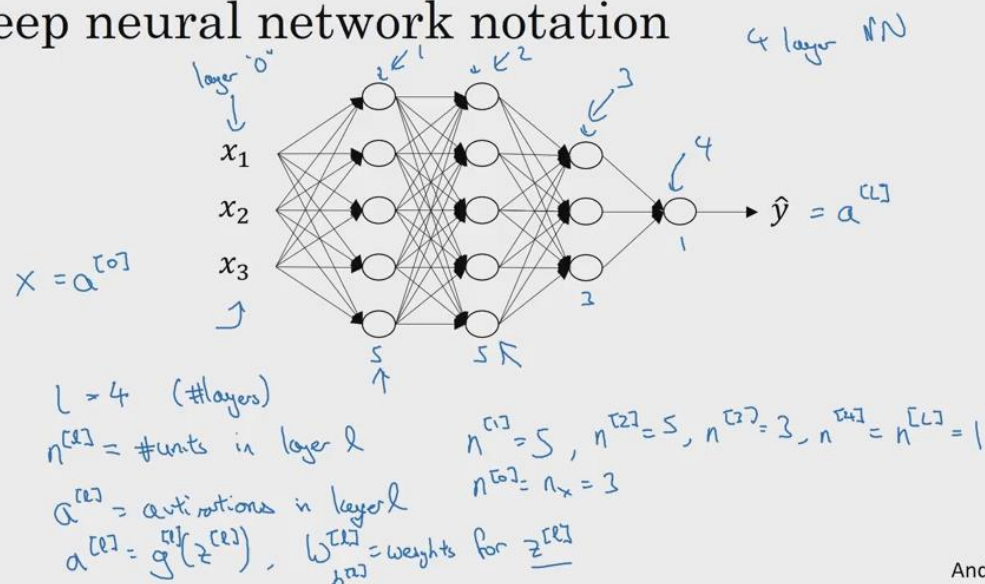
What is a deep neural network?



Andrew Ng

Figura 24 - Slide 1 do vídeo Deep L-layer neural network" da semana 4

Deep neural network notation



Andrew Ng

Figura 25 - Slide 1 do vídeo "Deep L-layer neural network" da semana 4

Forward propagation in a Deep Network

A propagação para frente, em questão de cálculos, continua a mesma:

- $z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$
- $A^{[l]} = g(z^{[l]})$

A única diferença entre as redes mais rasas e as redes mais profundas é que este cálculo é feito dentro de um loop.

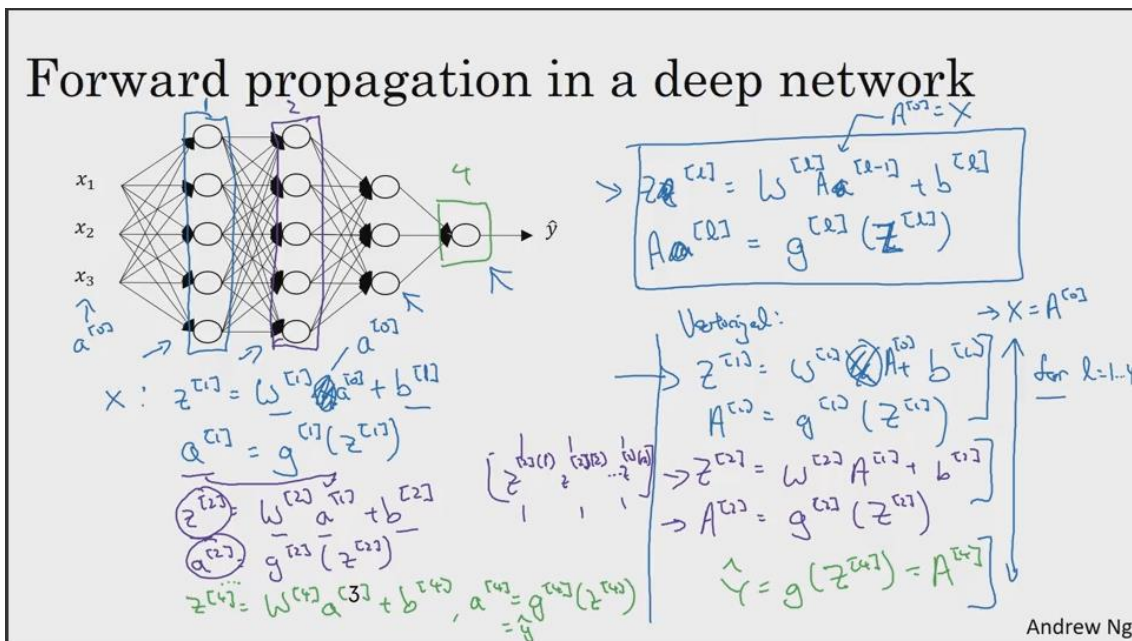


Figura 26 - Slide da aula "Forward propagation in a Deep Network" da semana 4

Getting your matrix dimensions right

Para criarmos uma rede neural, precisamos também saber mensurar o tamanho e mensurar quais serão os nós da rede. Como o cálculo é feito através de multiplicação de vetores, precisamos entender os conceitos de como isso é feito.

Na multiplicação de vetores, por exemplo $A \times B$, não sendo uma multiplicação elemento por elemento, as **colunas** de A deve ser iguais as **linhas** de B .

$$A = \begin{bmatrix} 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 11 \\ 2 & 23 \end{bmatrix} \quad np.dot(A, B) = \begin{bmatrix} 11 & 23 \end{bmatrix}$$

Podemos ver que o resultado será uma matriz com tamanho de **A pelo tamanho de B**.

"Mas qual a importância disso?" – É com isso que medimos como será nossa rede.

Partindo que teremos parâmetros **X** (neste caso serão 2 parâmetros) e três neurônios, a disposição será como descrito acima. **B** sendo os parâmetros **x** e **A** sendo o nosso **W**. Não esquecendo que cada coluna do **W** representa um parâmetro diferente. Isso quer dizer que **W[0][0]** é sobre o primeiro elemento de **X** e **W[0][1]** é sobre o segundo elemento de **X**. Interpretando isso em imagens, como os exemplos principais do curso, a segunda coluna de **W** é sobre **um pixel da segunda imagem**.

De uma forma resumida: o tamanho do seu **W** será a quantidade de neurônios da frente vezes a quantidade de neurônios anteriores.

Partindo do exemplo da imagem acima e tendo como exemplo uma implementação em loops, **z[0]** (o primeiro resultado do cálculo) tem que ser um vetor **5x1**, sendo que nós temos 3 inputs **X**, então temos que ter um **W** com **5x3** elementos.

Se tratando do **b**, ele terá que ter o tamanho do resultado de **Z**, ou seja, no exemplo acima teremos que ter um **b** com **5x1** elementos.

Já quando tratamos de uma implementação usando **multiplicação de vetores**, **Z** terá dimensões diferentes.

Sendo **Z** o resultado do cálculo $W^{[l]} X + b^{[l]}$, e **Z** possuindo **n** parâmetros (neste caso são as várias imagens inseridas), o resultado de **Z** terá o tamanho de $n^{[l]}, m$, sendo **m** a quantidade de parâmetros incluídos.

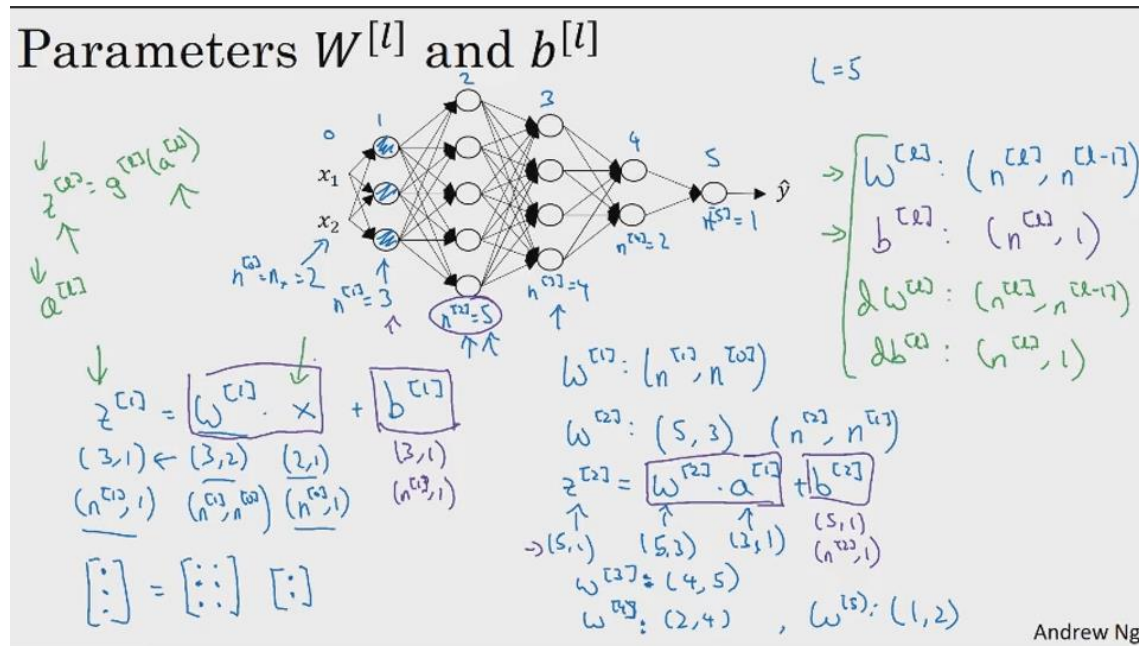


Figura 27- Slide 1 do vídeo "Getting your matrix dimensions right", onde o professor Ng explica sobre a implementação na forma de loops

Vectorized implementation

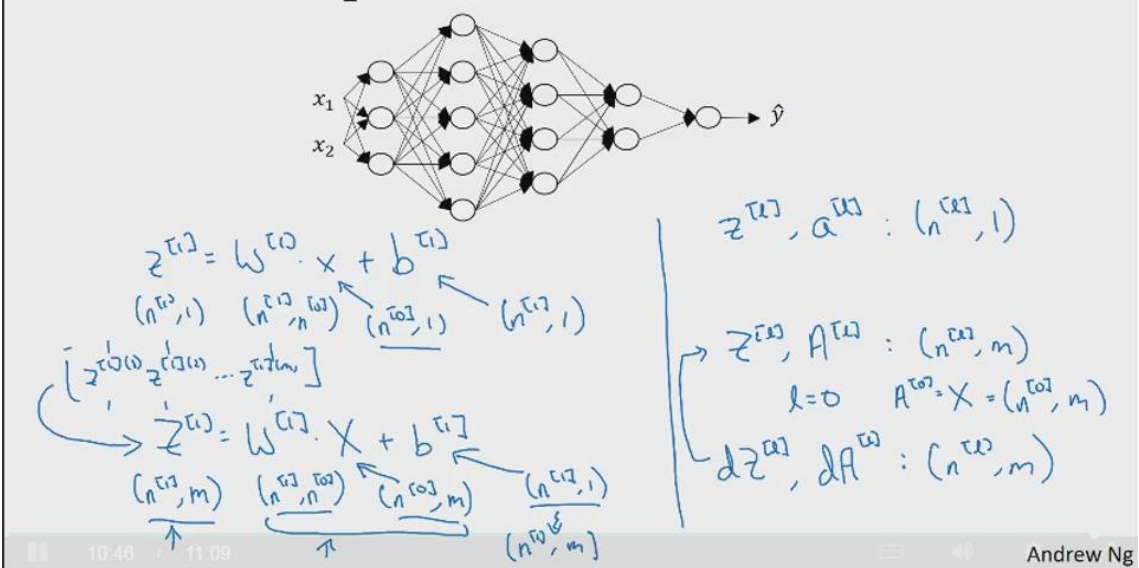


Figura 28 - Slide 2 do vídeo "Getting your matrix dimensions right", onde o professor Ng demonstra a implementação vetorizada

Parâmetros e Hiperparâmetros

Os parâmetros são o W e b , já os hiperparâmetros são aqueles que irão modificar ou dar o passo para W e b , tais quais o alfa, qual função de ativação, a quantidade de camadas e nós e assim por diante. Estes valores são os **hiperparâmetros**.

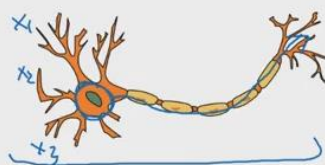
Forward and backward propagation

$$\begin{aligned} Z^{[1]} &= W^{[1]}X + b^{[1]} \\ A^{[1]} &= g^{[1]}(Z^{[1]}) \\ Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]} \\ A^{[2]} &= g^{[2]}(Z^{[2]}) \\ &\vdots \\ A^{[L]} &= g^{[L]}(Z^{[L]}) = \hat{Y} \end{aligned}$$

"It's like the brain"



$$\begin{aligned} dZ^{[L]} &= A^{[L]} - Y \\ dW^{[L]} &= \frac{1}{m} dZ^{[L]} A^{[L]T} \\ db^{[L]} &= \frac{1}{m} \text{np.sum}(dZ^{[L]}, \text{axis} = 1, \text{keepdims} = \text{True}) \\ dZ^{[L-1]} &= dW^{[L]T} dZ^{[L]} g'^{[L]}(Z^{[L-1]}) \\ &\vdots \\ dZ^{[1]} &= dW^{[L]T} dZ^{[2]} g'^{[1]}(Z^{[1]}) \\ dW^{[1]} &= \frac{1}{m} dZ^{[1]} A^{[1]T} \\ db^{[1]} &= \frac{1}{m} \text{np.sum}(dZ^{[1]}, \text{axis} = 1, \text{keepdims} = \text{True}) \end{aligned}$$



Andrew Ng

Figura 29 - Slide do vídeo "What does this have to do with the brain?"

Anotação sobre o cálculo dos vetores.

Usamos a vetorização das variáveis para que o cálculo seja executado de forma muito mais rápida do que usar loops. Pelos vídeos, o professor Ng deixa claro que os valores de entrada das imagens a serem analisadas são vetorizadas. Isso quer dizer que se uma imagem possuir 64px * 64px * 3 cores, cada pixel da imagem será colocado em um vetor, onde cada linha é um pixel (terá 64 * 64 * 3 linhas = 12.288 linhas no total).

Exemplo: Digamos que cada coluna seja uma imagem e cada valor seja um pixel:

0,1	0,4	0,7
0,2	0,5	0,8
0,3	0,6	0,9

Cada coluna representa uma imagem. Então [0.1,0.2,0.3] é a imagem A, [0.4,0.5,0.6] é a imagem B e [0.7,0.8,0.9] é a imagem C. **Usei o “.” para não confundir a vírgula do vetor com a vírgula do número.**

Como a fórmula é $\sigma(z)$, sendo $z = w^T x + b$, onde o **w** são os valores que serão usados para “predizer” a imagem, **x** é o vetor 3 x 3 acima representado e **b** é o número que também é usado como peso da função. Após o cálculo usando a função **.dot**, teremos o resultado **a**.

Por que precisamos de funções de ativação não lineares?

Sem usar uma função não linear, é a mesma coisa que simplesmente computarmos funções lineares, tal qual uma regressão linear ou, mesmo que usássemos os resultados destes, teríamos, por exemplo, somente uma simples regressão logística. Se for para computar funções lineares, é mais fácil fazê-las de forma simples.

Um lugar que podemos usar uma função linear de ativação é quando o resultado é um número real. Aí não podemos usar funções como a regressão logística nos passos anteriores ao valor predito, mas usamos **ReLU**, **Thana** ou outros tipo de funções não lineares.

COMANDOS EM PYTHON

.shape[x] – retorna a dimensão **x** do elemento selecionado. Objeto **numpy**

1 / (1 + np.exp(-x)) – Função Sigmoid

np.zeros((x,y)) – Cria uma matriz de zeros com **xy** dimensões. Colocar entre parênteses para interpretar que o x e y são o primeiro parâmetro

CURSO 2 - Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization

Train / Dev / Test sets

A questão da distribuição dos dados entre as três partes (treino, desenvolvimento e teste) varia de aplicação para aplicação. Em épocas onde o volume de dados era menor (ou em outros cursos de Machine Learning), era sugerido que distribuíssemos os dados em uma fração de 70%/15%/15%, ou 80%/10%/10% e assim por diante. Com a era do big data, onde o volume de dados pode passar dos gigas em poucos segundos, podemos utilizar de frações muito menores nas partições de teste e desenvolvimento, já que determinada porcentagem, dependendo do volume de dados aos quais você vai trabalhar, pode ser uma grande quantidade.

Bias / variance

Underfitting possui um baixo viés (low bias), pois generaliza de uma forma pobre;

Overfitting deixa o modelo quase que igual aos dados de treino, mas não generaliza muito bem;

Quando verificamos os algoritmos, temos que analisar a porcentagem de erro nos sets de treino e dev e também temos que comparar os resultados com o mundo real. Se um ser humano tem 10% de chance de errar em identificar uma imagem, se o algoritmo errar 11%, o algoritmo é 1% pior que um ser humano.

Basic recipe for Machine Learning

- Caso tenha um **alto viés**, considere aumentar sua rede neural e/ou aumentar o tempo de treino até que a rede se adapte aos dados;

Regularization