



Redes Neurais

Bruno Fernandes



Conteúdo

- Rede de duas camadas
- Funções não-lineares
- Inicialização dos pesos
- O que é uma rede neural profunda?
- Melhorando o processo de otimização
- Batch-normalization
- Classificação multiclasse
- Frameworks de deep learning

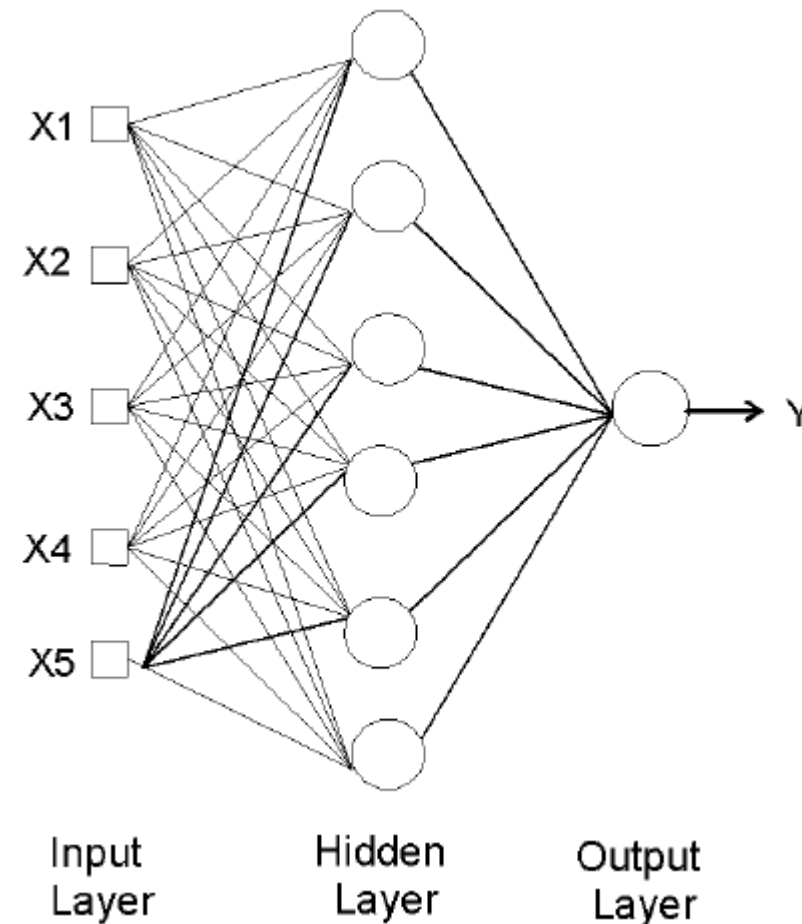


Conteúdo

- **Rede de duas camadas**
- Funções não-lineares
- Inicialização dos pesos
- O que é uma rede neural profunda?
- Melhorando o processo de otimização
- Batch-normalization
- Classificação multiclasse
- Frameworks de deep learning

Rede de duas camadas

- $z[i]$ -> somatório de entrada para a camada i utilizando $w[i]$, $a[i-1]$ e $b[i]$
- $a[i]$ -> ativação na camada i
- $a[0] = X$
- $a[2]$ -> saída da rede
- $z[i] = w[i]a[i-1] + b[i]$
- $a[i] = \sigma(z[i])$

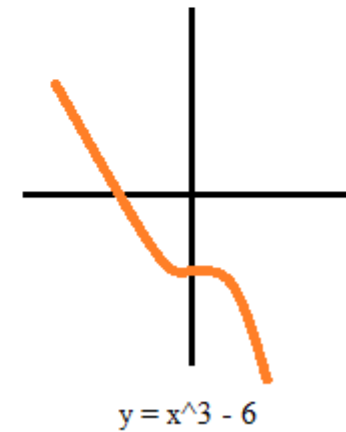
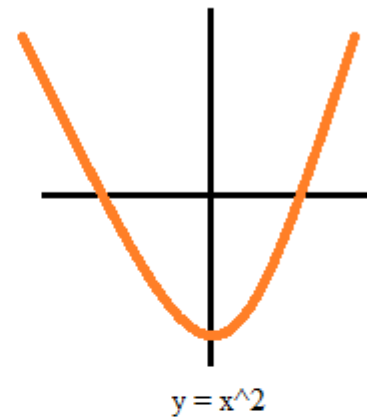
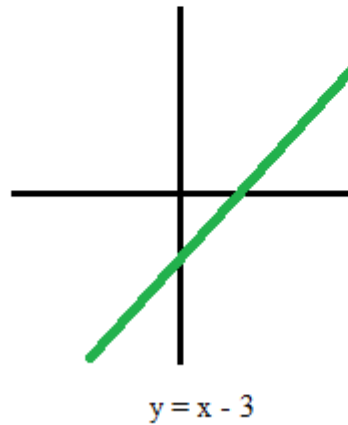
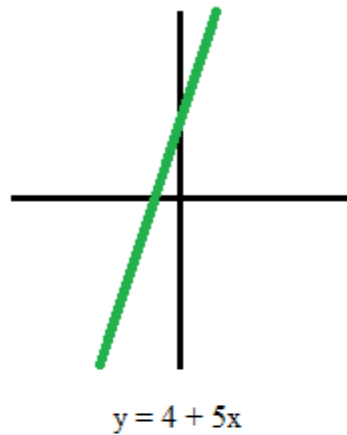




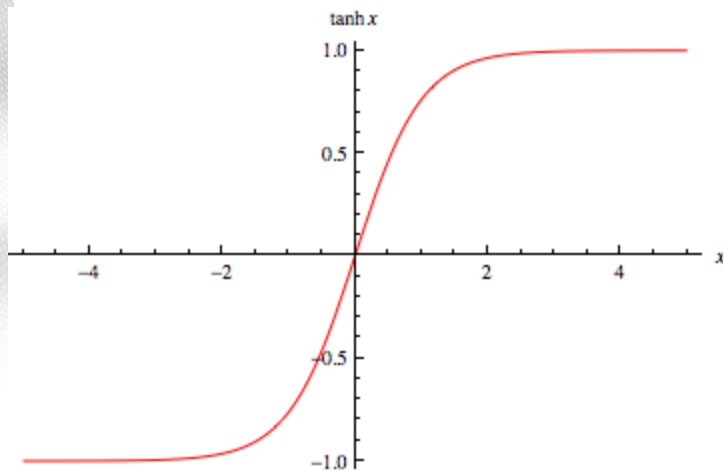
Conteúdo

- Rede de duas camadas
- **Funções não-lineares**
- Inicialização dos pesos
- O que é uma rede neural profunda?
- Melhorando o processo de otimização
- Batch-normalization
- Classificação multiclasse
- Frameworks de deep learning

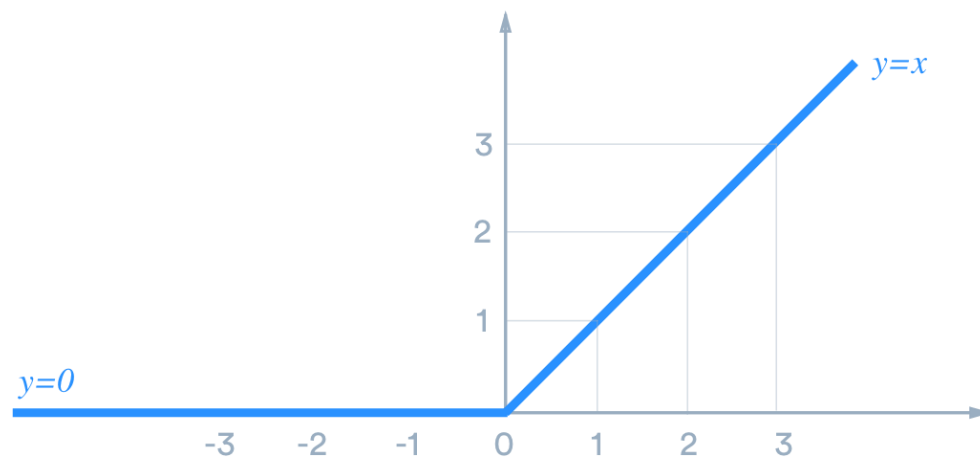
Funções não-lineares



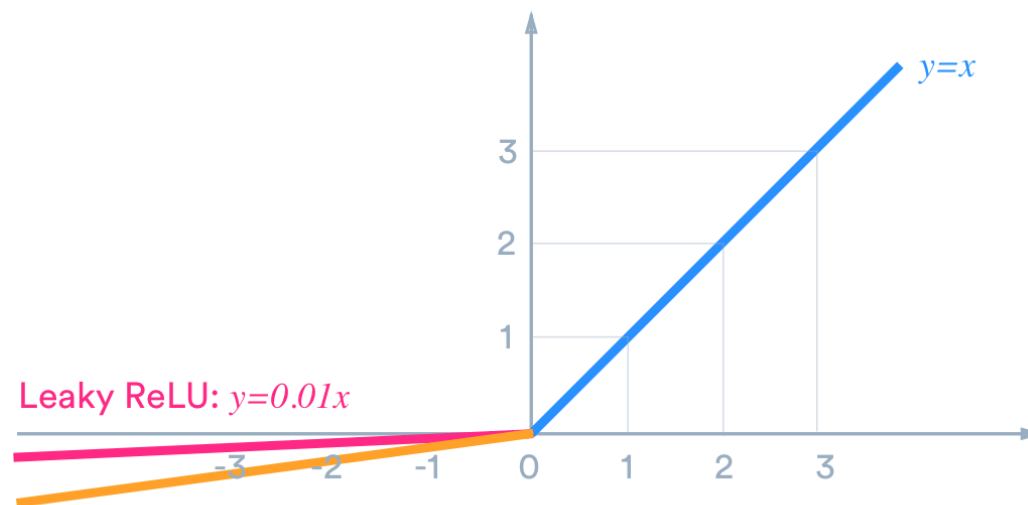
Funções não lineares



$$a = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



Rectified Linear Unit (ReLU) $\rightarrow a = \max(0, z)$



Parametric ReLU: $y=ax$

Leaky RELY $\rightarrow a = \max(0.01z, z)$



Por que precisamos de funções não-lineares?

- Do contrário, o somatório de várias funções lineares é o equivalente a computação de uma única função linear, sendo W e b a combinação linear de todos os W 's e b 's da rede
- A única indicação plausível para uma função linear seria em um problema de regressão com número reais na camada de saída, como na estimativa do preço de uma casa, onde a saída pode ser um número real entre R\$0,00 e R\$1.000.000,00
 - Todavia, ainda nesse caso seria melhor usar RELU



Conteúdo

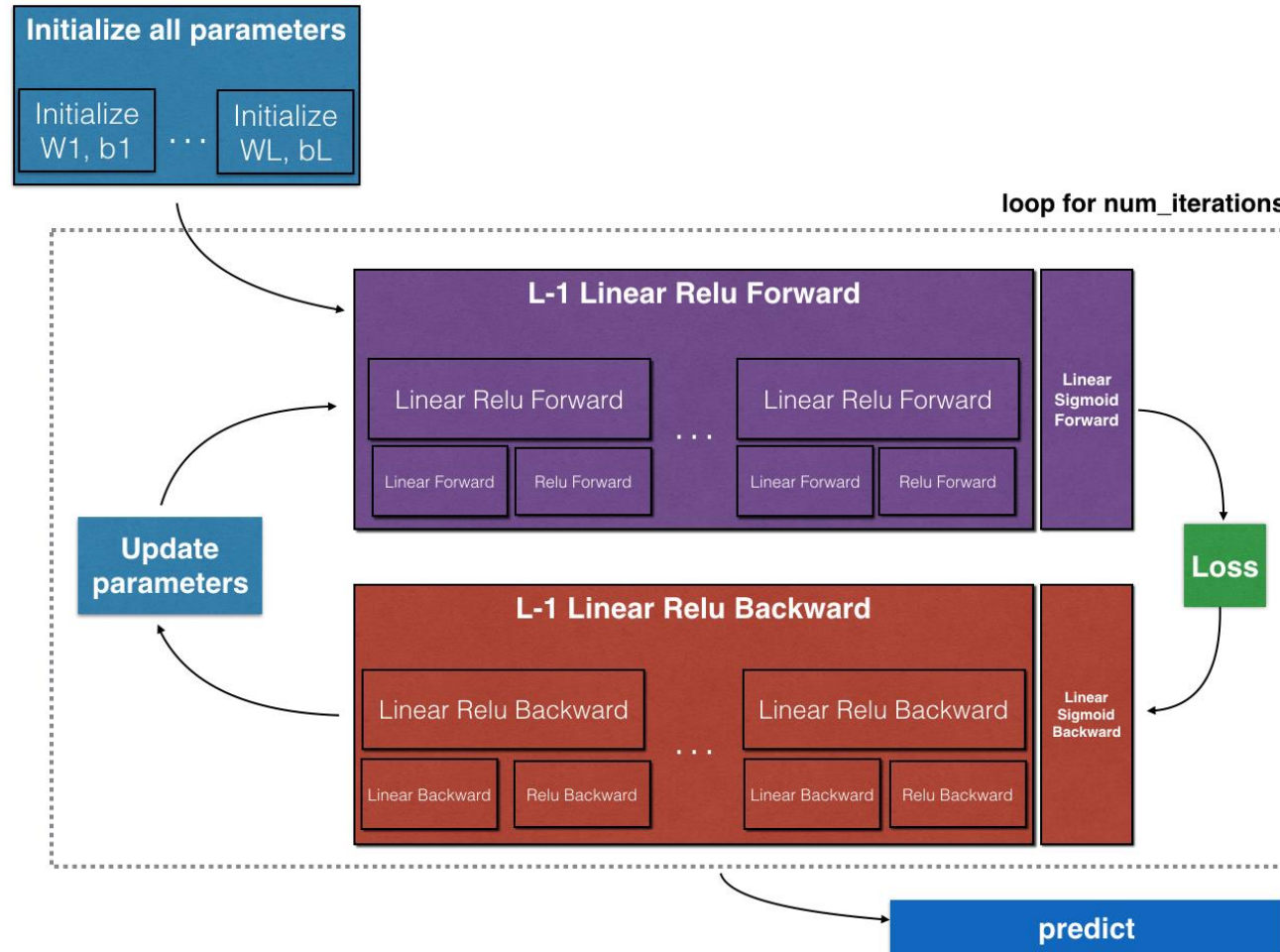
- Rede de duas camadas
- Funções não-lineares
- **Inicialização dos pesos**
- O que é uma rede neural profunda?
- Melhorando o processo de otimização
- Batch-normalization
- Classificação multiclasse
- Frameworks de deep learning



Inicialização dos pesos

- Se inicializar todos os pesos com 0, as unidades escondidas irão apresentar sempre as mesmas derivadas. Ou seja, não precisa de mais do que uma unidade escondida
- Solução: inicialização aleatória!
- Inicia-se com pesos pequenos porque se o valor for muito alto, pode terminar em um *plateau*
 - Local onde a função é plana e a derivada é próxima de 0 para unidades com função de ativação sigmoide ou tanh

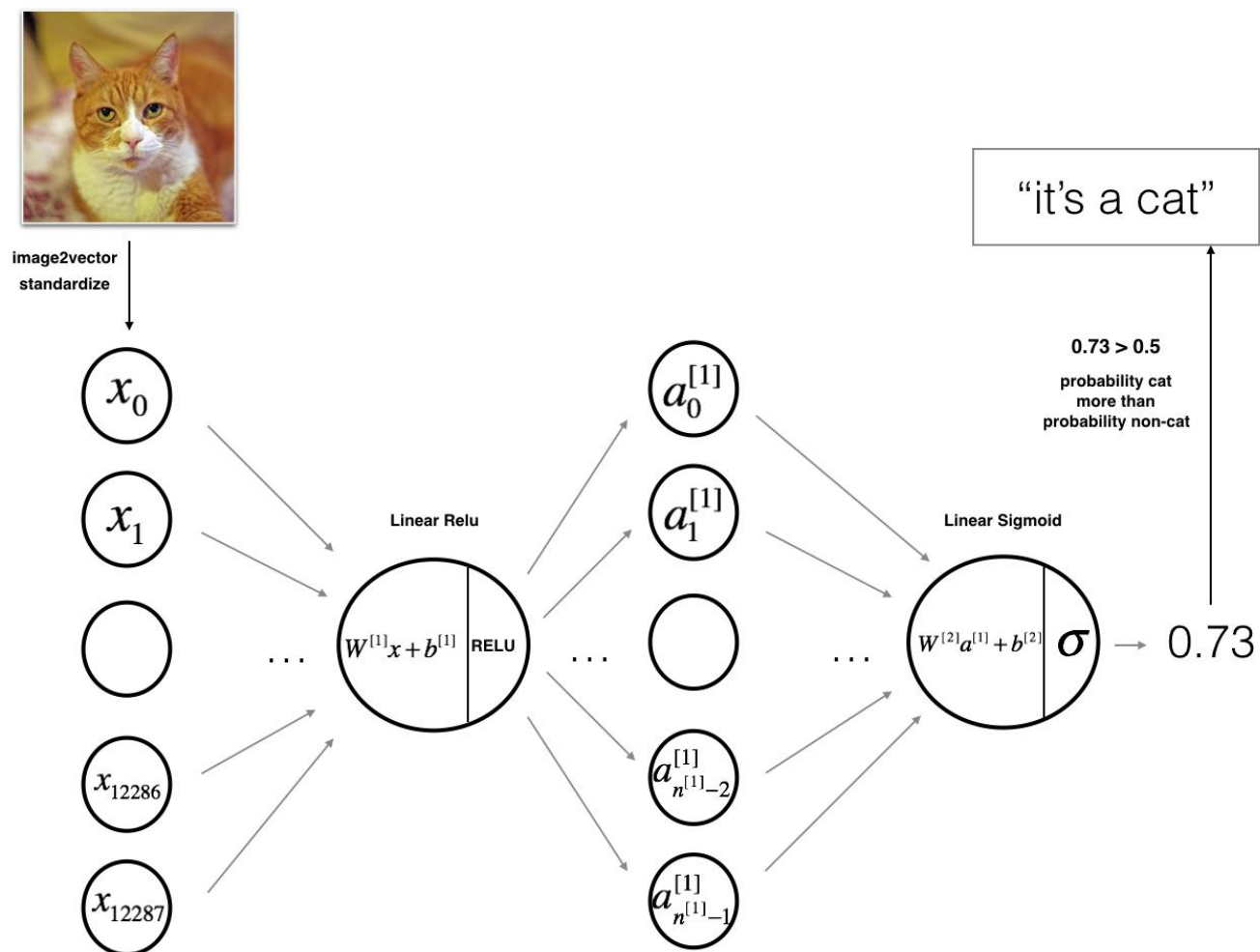
Exemplo de rede neural em Python



Andrew Ng, "Deep Learning Specialization". Coursera, 2018.



Rede Neural de duas camadas



Andrew Ng, "Deep Learning Specialization". Coursera, 2018.



Funções de ativação e suas derivadas

```
def sigmoid(Z):
```

```
    A = 1/(1+np.exp(-Z))
```

```
    cache = Z
```

```
    return A, cache
```

```
def relu(Z):
```

```
    A = np.maximum(0,Z)
```

```
    cache = Z
```

```
    return A, cache
```

```
def relu_backward(dA, cache):
```

```
    Z = cache
```

```
    dZ = np.array(dA, copy=True) dZ[Z <= 0] = 0
```

```
    return dZ
```

```
def sigmoid_backward(dA, cache):
```

```
    Z = cache
```

```
    s = 1/(1+np.exp(-Z))
```

```
    dZ = dA * s * (1-s)
```

```
    return dZ
```





Inicialização dos pesos


```
def initialize_parameters(n_x, n_h, n_y):  
    W1 = np.random.randn(n_h, n_x) * 0.01  
    b1 = np.zeros((n_h, 1))  
    W2 = np.random.randn(n_y, n_h) * 0.01  
    b2 = np.zeros((n_y, 1))  
  
    parameters = {"W1": W1,  
                  "b1": b1,  
                  "W2": W2,  
                  "b2": b2}  
  
    return parameters
```




Forward

```
def linear_forward(A, W, b):
    Z = np.dot(W, A) + b
    cache = (A, W, b)
    return Z, cache

def linear_activation_forward(A_prev, W, b, activation):
    if activation == "sigmoid":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)
    elif activation == "relu":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)
    cache = (linear_cache, activation_cache)
    return A, cache
```





Backward

```
def linear_backward(dZ, cache):
```

```
    A_prev, W, b = cache
```

```
    m = A_prev.shape[1]
```

```
    dW = 1/m * np.dot(dZ, A_prev.T)
```

```
    db = np.sum(dZ, axis=1, keepdims=True)/m
```

```
    dA_prev = np.dot(W.T, dZ)
```

```
    return dA_prev, dW, db
```

```
def linear_activation_backward(dA, cache, activation):
```

```
    linear_cache, activation_cache = cache
```

```
    if activation == "relu":
```

```
        dZ = relu_backward(dA, activation_cache)
```

```
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
```

```
    elif activation == "sigmoid":
```

```
        dZ = sigmoid_backward(dA, activation_cache)
```

```
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
```

```
    return dA_prev, dW, db
```




Atualização dos parâmetros

```
def update_parameters(parameters, grads, learning_rate):  
    L = len(parameters) // 2 # number of layers in the neural network  
    # Update rule for each parameter. Use a for loop.  
    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)]  
    return parameters
```



Cálculo do custo

```
def compute_cost(AL, Y):  
    m = Y.shape[1]  
    # Compute loss from aL and y.  
    cost = (-1/m) * np.sum( np.dot(Y, np.log(AL).T) + np.dot((1-Y), np.log(1-AL).T) )  
    cost = np.squeeze(cost)      # To make sure your cost's shape is what we expect (e.g.  
    this turns [[17]] into 17).  
    return cost
```

Modelo de duas camadas

```
def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations =  
3000, print_cost=False):  
    grads = {}  
    costs = [] # to keep track of the cost  
    m = X.shape[1] # number of examples  
    (n_x, n_h, n_y) = layers_dims  
    # Initialize parameters dictionary, by calling one of the functions you'd  
    # previously implemented  
    parameters = initialize_parameters(n_x, n_h, n_y)  
    # Get W1, b1, W2 and b2 from the dictionary parameters.  
    W1 = parameters["W1"]  
    b1 = parameters["b1"]  
    W2 = parameters["W2"]  
    b2 = parameters["b2"]
```



Modelo de duas camadas

```
def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000,
print_cost=False):
    # Loop (gradient descent)
    for i in range(0, num_iterations):
        # Forward propagation: LINEAR -> RELU -> LINEAR -> SIGMOID. Inputs: "X, W1, b1". Output:
        "A1, cache1, A2, cache2".
        A1, cache1 = linear_activation_forward(X, W1, b1, "relu")
        A2, cache2 = linear_activation_forward(A1, W2, b2, "sigmoid")
        # Compute cost
        cost = compute_cost(A2, Y)
        # Initializing backward propagation
        dA2 = - (np.divide(Y, A2) - np.divide(1 - Y, 1 - A2))
        # Backward propagation. Inputs: "dA2, cache2, cache1". Outputs: "dA1, dW2, db2; also dA0
        (not used), dW1, db1".
        dA1, dW2, db2 = linear_activation_backward(dA2, cache2, "sigmoid")
        dA0, dW1, db1 = linear_activation_backward(dA1, cache1, "relu")
```



Modelo de duas camadas

```
def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000,
print_cost=False):
    # Set grads['dW1'] to dW1, grads['db1'] to db1, grads['dW2'] to dW2, grads['db2'] to db2
    grads['dW1'] = dW1
    grads['db1'] = db1
    grads['dW2'] = dW2
    grads['db2'] = db2
    # Update parameters.
    parameters = update_parameters(parameters, grads, learning_rate)
    # Retrieve W1, b1, W2, b2 from parameters
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
```



Modelo de duas camadas

```
def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000, print_cost=False):  
    # Print the cost every 100 training example  
    if print_cost and i % 100 == 0:  
        print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))  
    if print_cost and i % 100 == 0:  
        costs.append(cost)  
  
    # plot the cost  
    plt.plot(np.squeeze(costs))  
    plt.ylabel('cost')  
    plt.xlabel('iterations (per tens)')  
    plt.title("Learning rate =" + str(learning_rate))  
    plt.show()  
    return parameters
```



Treinando o modelo

```
# Reshape the training and test examples
train_x_flatten = train_x_orig.reshape(train_x_orig.shape[0], -1).T # The "-1" makes reshape
# flatten the remaining dimensions
test_x_flatten = test_x_orig.reshape(test_x_orig.shape[0], -1).T
# Standardize data to have feature values between 0 and 1.
train_x = train_x_flatten/255.
test_x = test_x_flatten/255.
print ("train_x's shape: " + str(train_x.shape))
print ("test_x's shape: " + str(test_x.shape))
n_x = 12288 # num_px * num_px * 3
n_h = 7
n_y = 1
layers_dims = (n_x, n_h, n_y)
parameters = two_layer_model(train_x, train_y, layers_dims = (n_x, n_h, n_y), num_iterations =
2500, print_cost=True)
```





Predição

```
def model_forward(X, parameters):  
    A1, cache1 = linear_activation_forward(X, parameters["W1"],  
parameters["b1"], "relu")  
    A2, cache2 = linear_activation_forward(A1, parameters["W2"],  
parameters["b2"], activation = "sigmoid")  
    return A2
```




Predição

```
def predict(X, y, parameters):  
    m = X.shape[1]  
    n = len(parameters) // 2 # number of layers in the neural network  
    p = np.zeros((1,m))  
    # Forward propagation  
    probas = model_forward(X, parameters)  
    # convert probas to 0/1 predictions  
    for i in range(0, probas.shape[1]):  
        if probas[0,i] > 0.5:  
            p[0,i] = 1  
        else:  
            p[0,i] = 0  
    print("Accuracy: " + str(np.sum((p == y)/m)))  
    return p
```



Predição

```
predictions_train = predict(train_x, train_y,  
parameters)
```

```
predictions_test = predict(test_x, test_y,  
parameters)
```

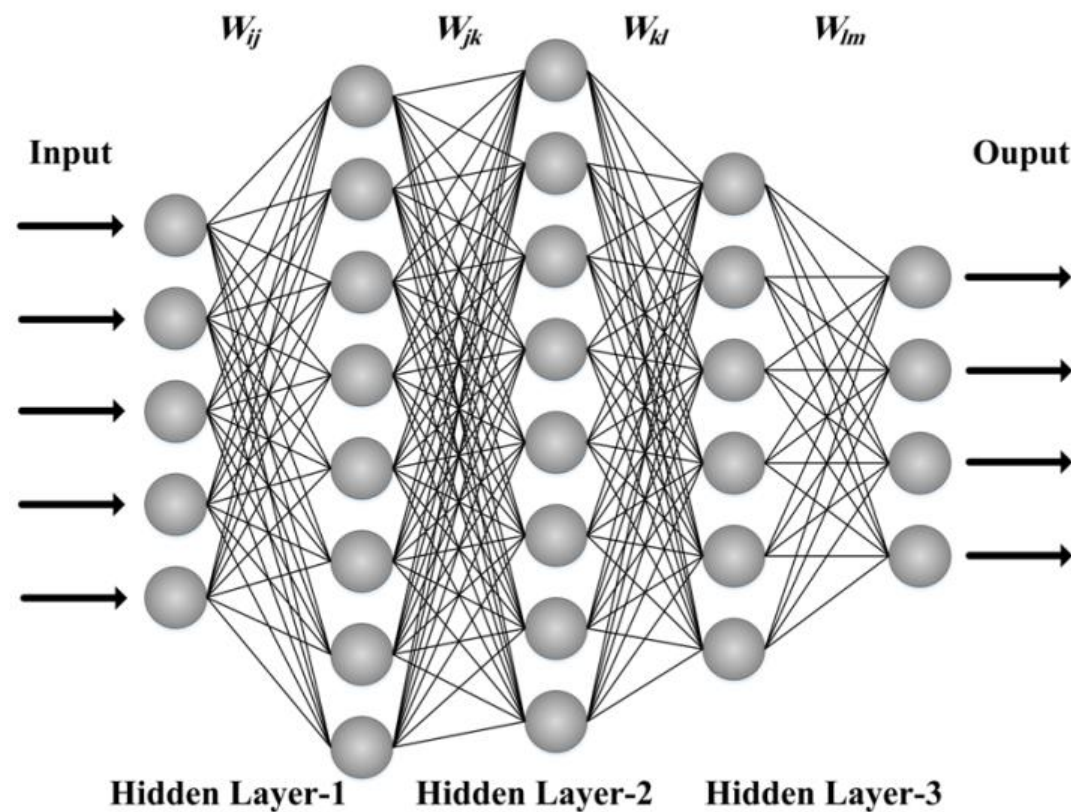


Conteúdo

- Rede de duas camadas
- Funções não-lineares
- Inicialização dos pesos
- **O que é uma rede neural profunda?**
- Melhorando o processo de otimização
- Batch-normalization
- Classificação multiclasse
- Frameworks de deep learning

O que é uma rede profunda?

- Em essência, quanto mais camadas escondidas mais profunda é uma rede

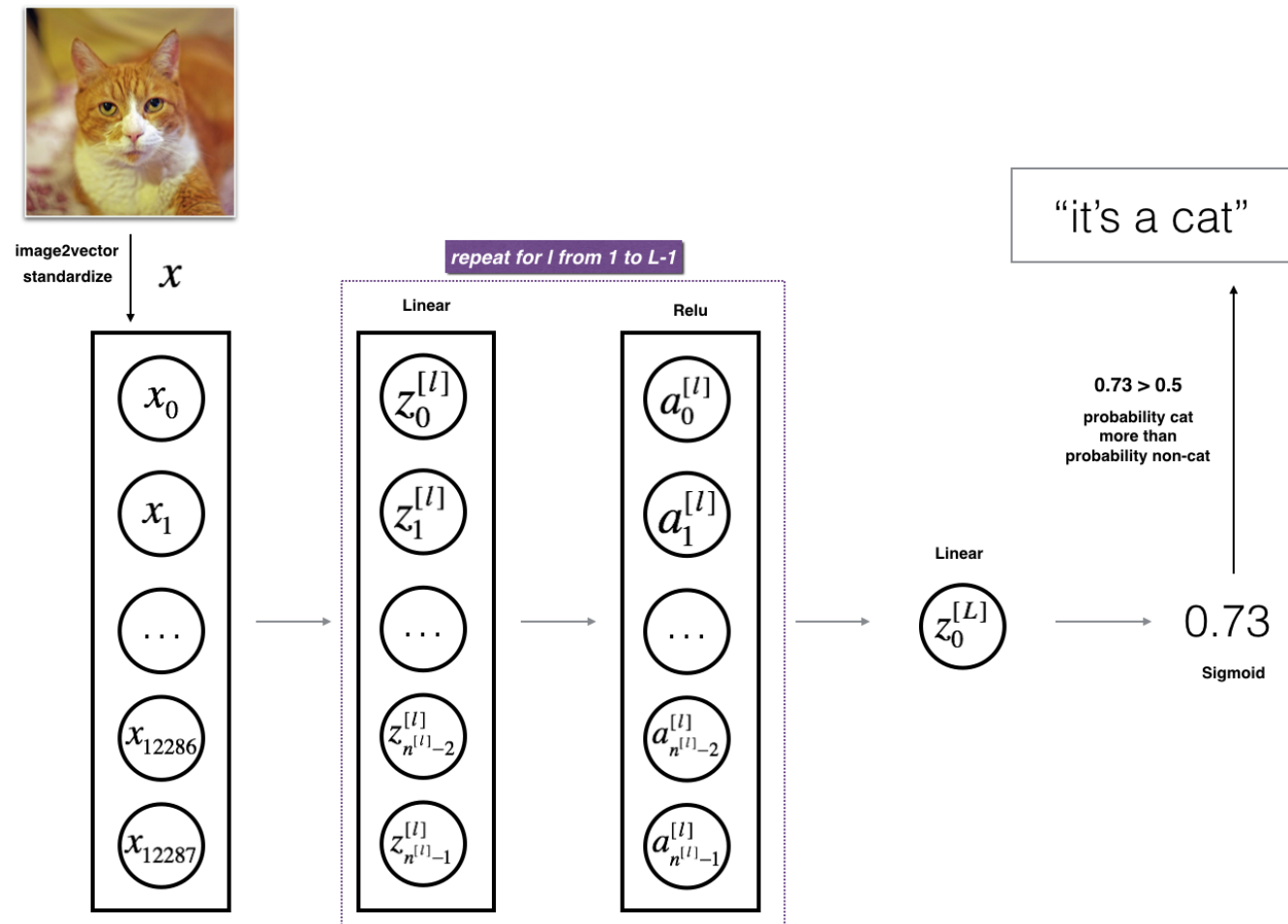




Por que representação profunda?

- Características simples são agrupadas para reconhecimento mais complexos
- Algumas funções podem ser computadas em redes neurais profundas com poucos neurônios enquanto as redes rasas precisam exponencialmente de mais neurônios
 - Ex: $x[1] \text{ XOR } x[2] \text{ XOR } x[3] \text{ XOR } \dots \text{ XOR } x[n]$ -> profunda é $O(\log n)$, rasa é $O(2^n)$

Rede Neural de L camadas



Andrew Ng, "Deep Learning Specialization". Coursera, 2018.





Implementação

- Mantem as seguintes funções
 - **sigmoid(Z)**
 - **relu(Z)**
 - **relu_backward(dA, cache)**
 - **sigmoid_backward(dA, cache)**
 - **load_dataset()**
 - **linear_forward(A, W, b)**
 - **linear_activation_forward(A_prev, W, b, activation)**
 - **L_model_forward(X, parameters)**
 - **compute_cost(AL, Y)**
 - **linear_backward(dZ, cache)**
 - **linear_activation_backward(dA, cache, activation)**
 - **update_parameters(parameters, grads, learning_rate)**
 - **predict(X, y, parameters)**



Inicialização dos pesos

```
def initialize_parameters_deep(layer_dims):  
    np.random.seed(1)  
    parameters = {}  
    L = len(layer_dims)           # number of layers in  
the network  
    for l in range(1, L):  
        parameters['W' + str(l)] =  
np.random.randn(layer_dims[l], layer_dims[l-1]) /  
np.sqrt(layer_dims[l-1]) #*0.01  
        parameters['b' + str(l)] =  
np.zeros((layer_dims[l], 1))  
    return parameters
```




Forward

```
def L_model_forward(X, parameters):  
    caches = []  
    A = X  
    L = len(parameters)  
    for l in range(1, L):  
        A_prev = A  
        A, cache = linear_activation_forward(A_prev, parameters["W" +  
str(L)], parameters["b" + str(L)], activation = "relu")  
        caches.append(cache)  
    AL, cache = linear_activation_forward(A, parameters["W"+str(L)],  
parameters["b"+str(L)], activation = "sigmoid")  
    caches.append(cache)  
    return AL, caches
```

Backward

```
def L_model_backward(AL, Y, caches):
    grads = {}
    L = len(caches) # the number of layers
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL
    dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))
    current_cache = caches[L-1]
    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] = linear_activation_backward(dAL,
    current_cache, "sigmoid")
    for l in reversed(range(L-1)):
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["dA" + str(l + 2)], current_cache,
        "relu")
        grads["dA" + str(l + 1)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp
    return grads
```



Modelo com L Camadas

```
def L_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000, print_cost=False):#lr was 0.009

    np.random.seed(1)

    costs = [] # keep track of cost

    parameters = initialize_parameters_deep(layers_dims)

    for i in range(0, num_iterations):

        AL, caches = L_model_forward(X, parameters)

        cost = compute_cost(AL, Y)

        grads = L_model_backward(AL, Y, caches)

        parameters = update_parameters(parameters, grads, learning_rate)

        if print_cost and i % 100 == 0:

            print ("Cost after iteration %i: %f" %(i, cost))

        if print_cost and i % 100 == 0:

            costs.append(cost)

    plt.plot(np.squeeze(costs))

    plt.ylabel('cost')

    plt.xlabel('iterations (per tens)')

    plt.title("Learning rate = " + str(learning_rate))

    plt.show()

    return parameters
```



Treinando e testando

```
train_x_orig, train_y, test_x_orig, test_y, classes =  
load_dataset()  
train_x_flatten = train_x_orig.reshape(train_x_orig.shape[0], -1).T  
test_x_flatten = test_x_orig.reshape(test_x_orig.shape[0], -1).T  
train_x = train_x_flatten/255.  
test_x = test_x_flatten/255.  
layers_dims = [12288, 20, 7, 5, 1] # 5-layer model  
parameters = L_layer_model(train_x, train_y, layers_dims,  
num_iterations = 2500, print_cost = True)  
predictions_train = predict(train_x, train_y, parameters)  
predictions_test = predict(test_x, test_y, parameters)
```





Regularização

- Ajuda a evitar o overfitting através da adição do somatório das normas das matrizes de pesos de todas as camadas multiplicada por $\frac{\lambda}{2m}$ (regularização L2)
- Na prática, isso é o equivalente a multiplicar o $w[l]$ durante o processo de ajuste dos pesos por $1 - \frac{\alpha\lambda}{2m}$
- Ou seja, $w^{[l]} := w^{[l]} - \alpha \frac{dJ(w,b)}{dw}$, passa a ser $w^{[l]} := w^{[l]} - \frac{\alpha\lambda}{2m} w^{[l]} - \alpha \frac{dJ(w,b)}{dw}$
- Dessa forma, um valor muito grande de λ faz w se aproximar de 0 (underfitting), enquanto um valor muito próximo a 0, permanece o overfitting



Dropout

- A cada iteração de treino descarta aleatoriamente alguns neurônios da rede
- Para isso, basta pegar os valores da ativação de alguns neurônios de uma determinada camada e fazê-los igual a zero
- Recomenda-se que após o processo divida as ativações restantes pelo valor da probabilidade de um neurônio não ser descartado

- Ex:

```
keep_prob=0.8
```

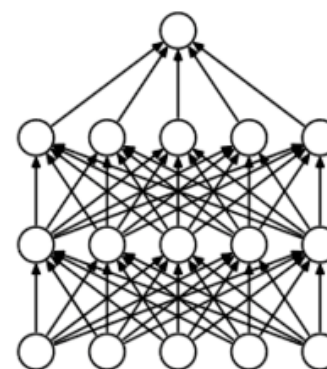
```
d3=np.random.randn(a3.shape[0],a3.shape[1]) < keep_prob
```

```
a3=a3*d3
```

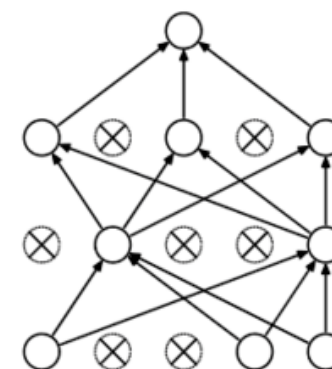
```
a3=a3/keep_prob
```


Dropout

- Dropout não é aplicado na fase de teste
- Dropout evita que uma única característica seja reforçada, uma vez que em iterações aleatórias, tal unidade simplesmente não emitirá qualquer sinal, pois terá sofrido o dropout
- Visão computacional usa muito o dropout devido ao grande tamanho dos padrões de entrada



(a) Standard Neural Net

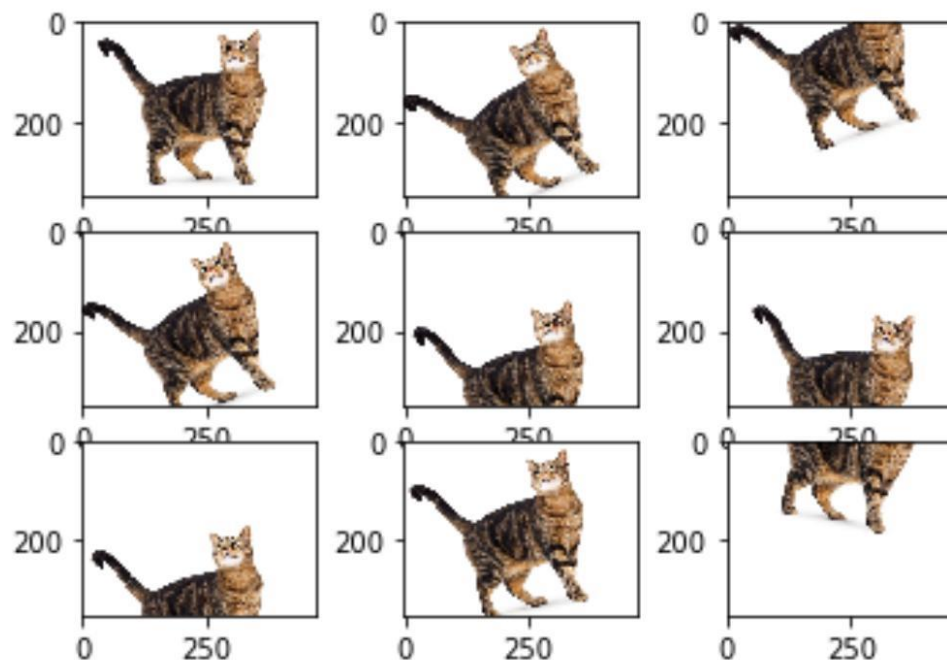


(b) After applying dropout.



Outras técnicas de regularização

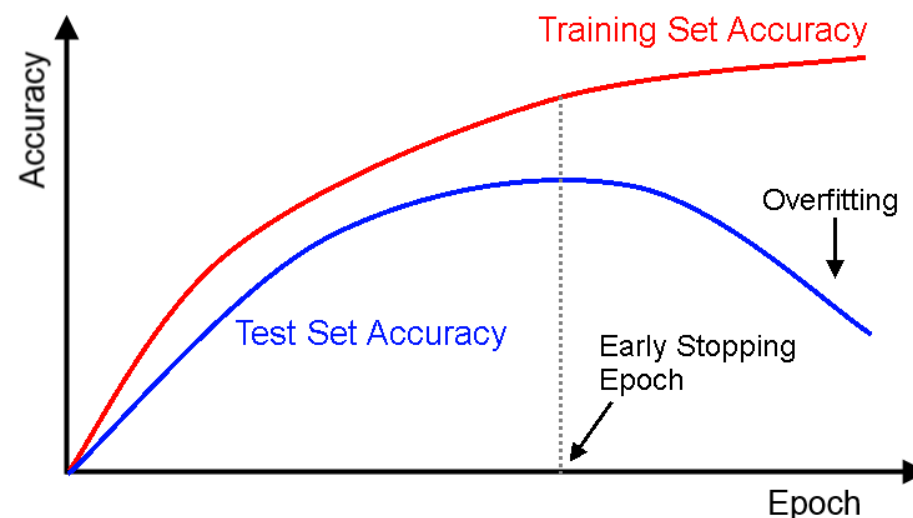
- Data augmentation



Outras técnicas de regularização

- Early stopping

- Não é uma boa escolha por acoplar os conceitos de otimizar função de custo e não sobreajustar ao mesmo tempo
- A vantagem é que mais fácil de verificar do que testar, por exemplo, vários valores de λ na regularização L2





Normalização

- Provê simetria para a função de custo J , ao invés de deixar a função de custo mais alongada e com a possibilidade de requerer muitos passos no processo de convergência caso a inicialização aleatória caia num extremo da superfície alongada
- Com as entradas normalizadas, gradiente descendente anda mais rápido
- Além disso, garante a mesma escala para todas as características
- Não apresenta desvantagens



Processo de normalização

- Subtraia a média (assim a média fica zero)
- Divida pela variância
- As mesmas médias e variâncias calculadas no treino são usadas nos dados de teste



Inicialização

- Ajuda a evitar os problemas do *vanishing or exploding gradients*
 - Acontece em redes profundas quando valores acima de 1 são acumulados, a saída vai crescer exponencialmente, explodindo os gradientes
 - Se forem menor que 1, vai tender a 0 -> *vanishing*
- Métodos de inicialização

$w^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{1}{n^{[l-1]}}\right)$, no caso de tanh (Xavier)

$w^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{2}{n^{[l-1]}}\right)$, no caso de relu

$w^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{2}{n^{[l-1]} + n^{[l]}}\right)$



Conteúdo

- Rede de duas camadas
- Funções não-lineares
- Inicialização dos pesos
- O que é uma rede neural profunda?
- **Melhorando o processo de otimização**
- Batch-normalization
- Classificação multiclasse
- Frameworks de deep learning



Melhorando o processo de otimização

- Maneiras de acelerar a convergência
 - Minibatch
 - Médias exponencialmente ponderadas
 - GD com momento
 - RMSProp
 - Adam
 - Decaimento da taxa de aprendizagem



Minibatch

- Serve para fazer as épocas de treino andarem mais rápido e facilitar a visualização da evolução de custo
 - Se o tamanho do minibatch = 1, é o gradiente descendente estocástico
 - Se o tamanho do minibatch = m , é o gradiente descendente em batch (Batch GD)
- Se o conjunto de treino é pequeno ($m \leq 2000$), use batch GD
- Valores típicos de minibatch: 64, 128, 256, 512, 1024 (esse último é mais raro)
- Faça de forma a caber em sua CPU/GPU



Médias exponencialmente ponderadas

- $v_t = \beta v_{t-1} + (1 - \beta)\theta_t$
- $v_0 = 0$
- θ_t seria o valor calculado para aquele dado momento
- Equivalente a média dos últimos $\frac{1}{1-\beta}$ valores no intervalo
- Existe um problema inicial por fazer $v_0 = 0$, a curva é puxada para baixo no início
- Ao invés de v_t , deve ser usado $\frac{v_t}{1-\beta^t}$, sendo que β^t tende a 0 com valor grande de t, uma vez que $\beta < 1$ (correção do bias)



GD com momento

- Solução para evitar o problema da taxa de aprendizagem muita baixa tornar a convergência lenta, enquanto a muito alta leva a divergência
- Solução

$$V_{dw} := \beta V_{dw} + (1 - \beta) dW$$

$$V_{db} := \beta V_{db} + (1 - \beta) db$$

$$W := W - \alpha V_{dw}$$

$$b := b - \alpha V_{db}$$

- $\beta=0.9$
- Normalmente, não se usa correção do bias para isso



RMSProp

- Ajuda a desacelerar atualizações que estão muito rápidas

$$S_{dw} := \beta S_{dw} + (1 - \beta) dW^2$$

$$S_{db} := \beta S_{db} + (1 - \beta) db^2$$

$$W := W - \alpha \frac{dw}{\sqrt{S_{dw}} + \varepsilon}$$

$$b := b - \alpha \frac{db}{\sqrt{S_{db}} + \varepsilon}$$

- Sugestão para $\varepsilon = 10^{-8}$



Adam

- Adaptive moment estimation
- A cada iteração de treino t
 - Computa dw e db com o minibatch corrente
 - Calcula V_{dw} e V_{db} com momento (usa-se parâmetro β_1)
 - Calcula S_{dw} e S_{db} com RMSProp (usa-se parâmetro β_2)
 - Faz a correção do bias
 - $V_{dw}^{corrected} = \frac{V_{dw}}{(1-\beta_1^t)}$, $V_{db}^{corrected} = \frac{V_{db}}{(1-\beta_1^t)}$, $S_{dw}^{corrected} = \frac{S_{dw}}{(1-\beta_2^t)}$, $S_{db}^{corrected} = \frac{S_{db}}{(1-\beta_2^t)}$
- $W := W - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected} + \epsilon}}$, $b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected} + \epsilon}}$



Adam

- Hiperparâmetros
 - α deve ser tunado
 - $\beta_1 = 0.9$
 - $\beta_2 = 0.999$
 - $\varepsilon = 10^{-8}$



Decaimento da taxa de aprendizagem

- A ideia é diminuir a velocidade do aprendizado enquanto se aproxima do mínimo

- $\alpha := \frac{1}{1+decay-rate*epoch-num} \alpha_0$

- $\alpha := 0.95^{epoch-num} \cdot \alpha_0$

- $\alpha := \frac{\kappa}{\sqrt{epoch-num}} \alpha_0$, κ é um parâmetro



Conteúdo

- Rede de duas camadas
- Funções não-lineares
- Inicialização dos pesos
- O que é uma rede neural profunda?
- Melhorando o processo de otimização
- **Batch-normalization**
- Classificação multiclasse
- Frameworks de deep learning



Batch-normalization

- Relembrando: normalização das entradas acelera convergência
- Para deep nets: podemos normalizar as ativações das camadas para treinar os pesos mais rapidamente?
 - Sim. Na prática, normaliza-se o somatório de entrada
 - Calcula a média e a variância de todas as entradas de uma camada e normaliza as entradas com tais valores
 - A entrada normalizada é ainda multiplicada por dois parâmetros aprendíveis para permitir que o próprio processo de backpropagation regule a influência da normalização (esses novos parâmetros possuem a mesma dimensão do bias)



Batch-normalization

- Como a média é subtraída dos dados, qualquer valor constante adicionado aos mesmo será desnecessário, logo o bias é desnecessário, seu papel é assumido por um dos parâmetros do batch-norm
- Vantagens
 - Torna o algoritmo mais invariante a ajustes na distribuição de entrada
 - A convergência é mais rápida
 - Batch-norm + minibatch adiciona algum ruído nas unidades de entrada, dando um efeito de regularização. Entretanto, não use batch-norm com o propósito de fazer regularização
- Normalmente, se resume a uma linha de código em frameworks de DL

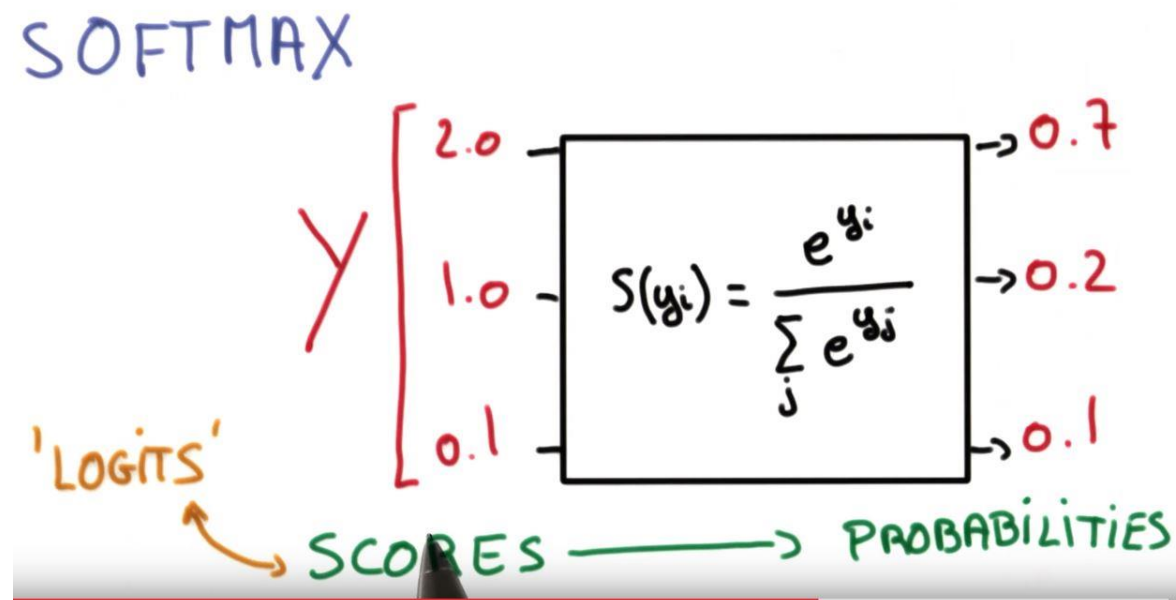


Conteúdo

- Rede de duas camadas
- Funções não-lineares
- Inicialização dos pesos
- O que é uma rede neural profunda?
- Melhorando o processo de otimização
- Batch-normalization
- **Classificação multiclasse**
- Frameworks de deep learning

Classificação multiclasse

- Utiliza-se a regressão softmax
- Cada neurônio na saída estima a probabilidade para uma classe
- Se a quantidade de neurônios na saída for igual a 2, softmax é reduzida a regressão logística





Conteúdo

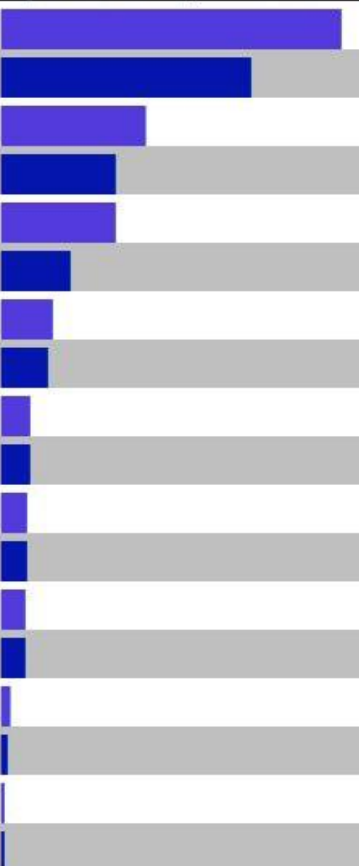
- Rede de duas camadas
- Funções não-lineares
- Inicialização dos pesos
- O que é uma rede neural profunda?
- Melhorando o processo de otimização
- Batch-normalization
- Classificação multiclasse
- **Frameworks de deep learning**



Frameworks de deep learning

- Caffe/Caffe2
 - CNTK
 - DL4J
 - Keras
 - Lasagne
 - MXNet
 - PaddlePaddle
 - **Tensorflow**
 - Theano
 - Torch
- Critérios para escolha
 - Facilidade de programação
 - Velocidade de execução
 - Aberto (código com boa governança)

Frameworks de deep learning

Aggregate popularity (30•contrib + 10•issues + 5•forks)•1e-3			
#1:	97.53		tensorflow/tensorflow
#2:	71.11		BVLC/caffe
#3:	43.70		fchollet/keras
#4:	32.07		Theano/Theano
#5:	31.96		dmlc/mxnet
#6:	19.51		deeplearning4j/deeplearning4j
#7:	15.63		Microsoft/CNTK
#8:	13.90		torch/torch7
#9:	9.03		pfnet/chainer
#10:	8.75		Lasagne/Lasagne
#11:	7.84		NVIDIA/DIGITS
#12:	7.83		mila-udem/blocks
#13:	5.95		karpathy/convnetjs
#14:	5.84		NervanaSystems/neon
#15:	4.91		tflearn/tflearn
#16:	3.28		amznlabs/amazon-dsstne
#17:	1.81		IDSIA/brainstorm
#18:	1.38		torchnet/torchnet





Exercício

- Implemente alguma das otimizações vistas no modelo de rede multicamadas



Redes Neurais

Bruno Fernandes