

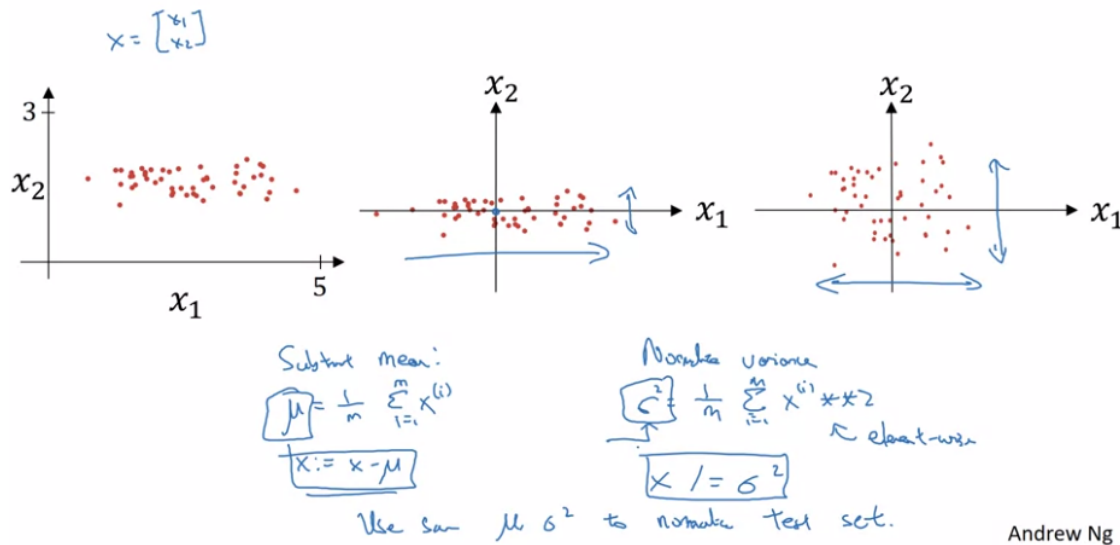
Week 1-3 Problem Set-up (Normalization, Initialization, G checking)

笔记本: DL 2 - Deep NN Hyperparameter Tuning, Regularization & Optimization

创建时间: 2021/1/9 09:43

更新时间: 2021/1/9 10:17

Normalizing training sets

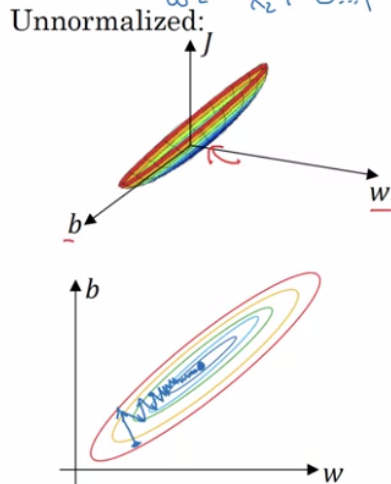


also do with test set, (divide by
sigma rather than sigma²)

Why normalize?

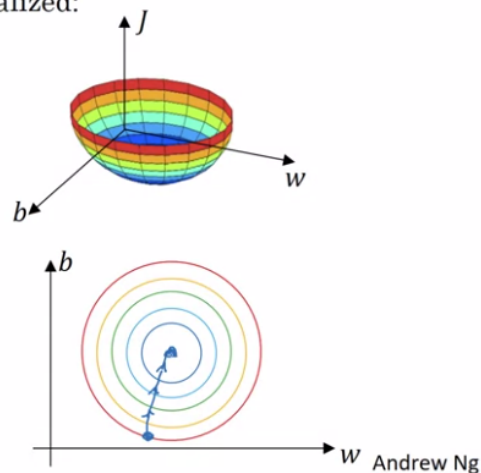
Why normalize inputs?

Unnormalized:
 $w_1, x_1: 1 \dots 1000$
 $w_2, x_2: 0 \dots 1$



$$J(w, b) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

Normalized:

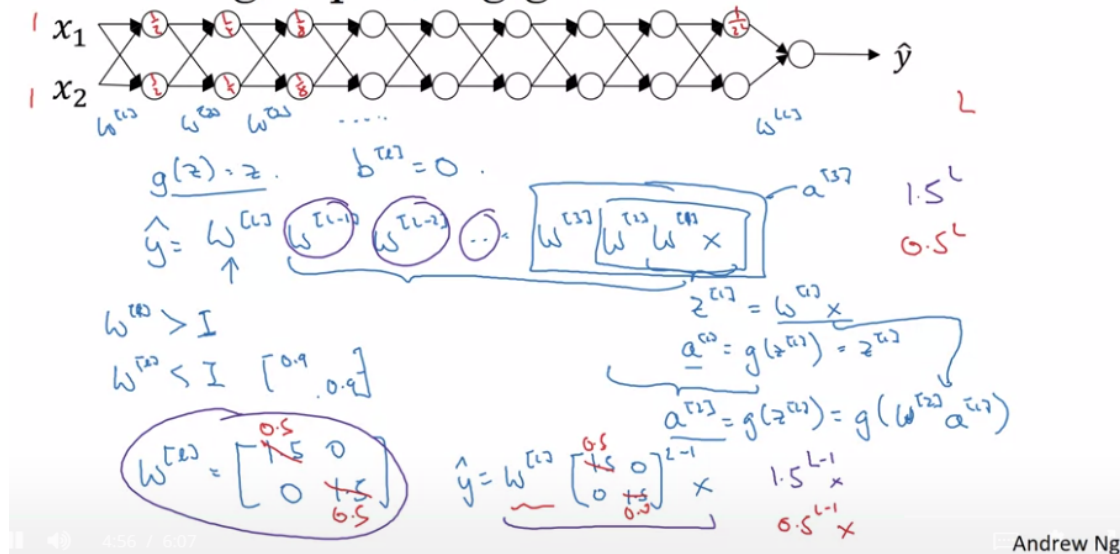


if you're running gradient descent on the cost function like the one on the left, then you might have to use a very **small learning rate** because if you're here that gradient descent might need a lot of steps to oscillate back and forth before it finally finds its way to the minimum. Whereas if you have a more spherical contours, then wherever you start gradient descent can pretty much go straight to the minimum. You can take much **larger steps** with gradient descent rather than needing to oscillate

around like the picture on the left.

Vanishing / Exploding Gradients

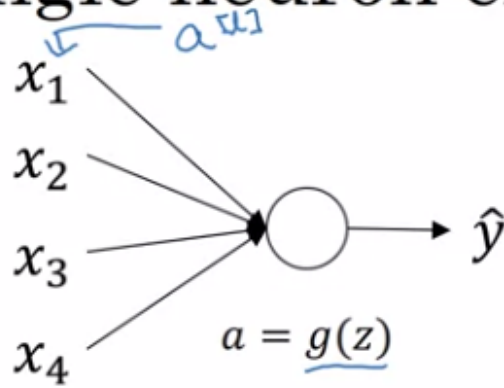
Vanishing/exploding gradients



(dW will also have such extreme problems)

Weight Initialization for NN

Single neuron example



$$z = \underbrace{w_1 x_1 + w_2 x_2 + \dots + w_n x_n}_{\text{large } n \rightarrow \text{smaller } w_i}$$

$$\text{Var}(w_i) = \frac{1}{n} \frac{2}{n}$$

$$w^{[1]} = \underbrace{n.p. \text{ random.}}_{\text{ReLU}} \cdot \underbrace{\text{random.}}_{g^{[1]}(z) = \text{ReLU}(z)} \cdot \underbrace{n.p. \text{ sqrt.}}_{\left(\frac{2}{n^{(1-1)}} \right)}$$

ReLU: He Initialization, $\text{sqrt}(2/n^{(l-1)})$

Xavier Initialization for tanh

Other variants:

$$\frac{1}{n^{(l-1)}}$$

tanh

Xavier initialization

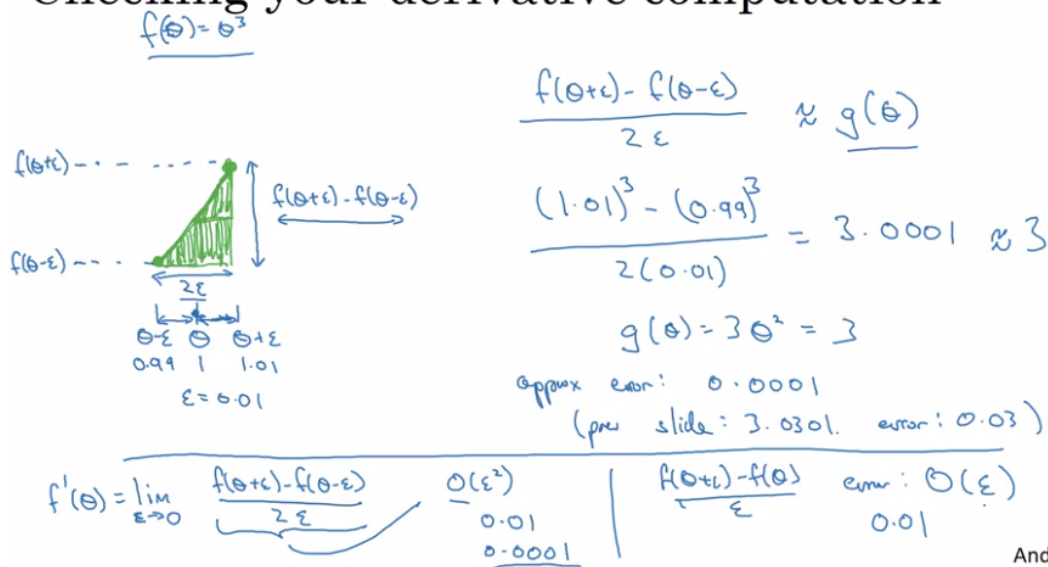
Let's derive Xavier Initialization now, step by step.

Our full derivation gives us the following initialization rule, which we apply to all weights:

$$W_{i,j}^{[l]} = \mathcal{N}\left(0, \frac{1}{n^{[l-1]}}\right)$$

Gradient Checking

Checking your derivative computation



Gradient check for a neural network

Take $W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}$ and reshape into a big vector θ .

concatenate

$$J(w^{(1)}, b^{(1)}, \dots, w^{(L)}, b^{(L)}) = J(\theta)$$

Take $dW^{[1]}, db^{[1]}, \dots, dW^{[L]}, db^{[L]}$ and reshape into a big vector $d\theta$.

Is $d\theta$ the gradient of $J(\theta)$?

Gradient checking (Grad check)

$$J(\theta) = J(\theta_1, \theta_2, \dots)$$

for each i :

$$\rightarrow \underline{d\theta_{approx}[i]} = \frac{J(\theta_1, \theta_2, \dots, \theta_i + \epsilon, \dots) - J(\theta_1, \theta_2, \dots, \theta_i - \epsilon, \dots)}{2\epsilon}$$

$$\approx \underline{d\theta[i]} = \frac{\partial J}{\partial \theta_i} \quad \Bigg| \quad d\theta_{approx} \approx d\theta$$

Check $\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2} \approx \frac{10^{-7}}{10^{-5}} = 10^{-3} - \text{wrong!}$

$\epsilon = 10^{-7} \rightarrow 10^{-3} - \text{wrong!}$

$10^{-7} \rightarrow \text{great} / 10^{-5} / 10^{-3} \rightarrow \text{wrong}$

Some other notes:

turn-off GC in training (only do this in debugging)

Gradient checking implementation notes

- Don't use in training – only to debug

$$\frac{d\theta_{approx}[i]}{\uparrow \uparrow} \leftrightarrow \frac{d\theta[i]}{\uparrow}$$

- If algorithm fails grad check, look at components to try to identify bug.

$$\underline{db^{(2)}} \quad \underline{dw^{(2)}}$$

- Remember regularization.

$$J(\theta) = \frac{1}{n} \sum_i f(y^{(i)}, \theta) + \frac{\lambda}{2n} \sum_i \|w^{(2)}\|_2^2$$

$d\theta = \text{gradient of } J \text{ wrt. } \theta$

- Doesn't work with dropout.

$$J \quad \underline{\text{keep-prob} = 1.0}$$

- Run at random initialization; perhaps again after some training.

$$\underline{w, b \approx 0}$$

Andrew Ng

it's not impossible that your implementation of gradient descent

is **correct when w and b are close to 0**, so at random initialization. But that as you run gradient descent and w and b become bigger, maybe your implementation of backprop is correct only when w and b is close to 0, but it gets more inaccurate when w and b become large. So one thing you could do, I don't do this very often, but one thing you could do is run grad check **at random initialization and then train the network for a while so that w and b have some time to wander away from 0**, from your small random initial values. And then run grad check again after you've trained for some number of iterations.

