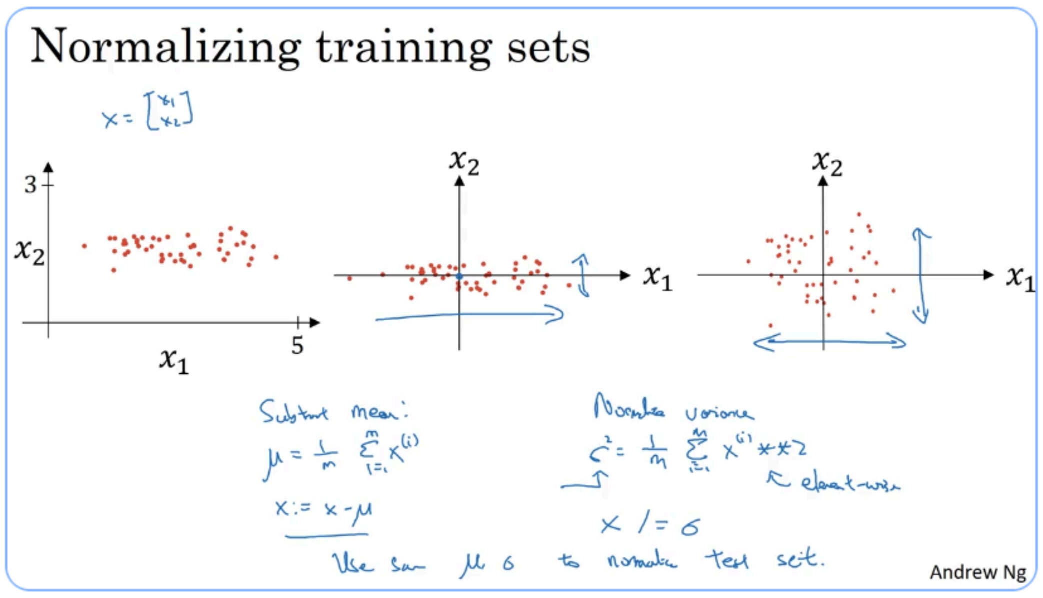
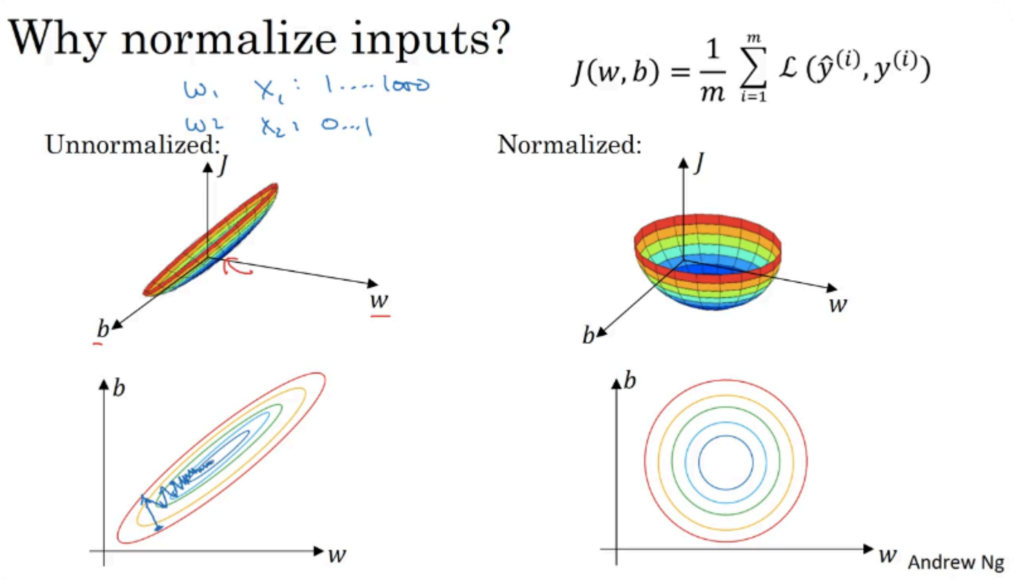
Setting Up your Optimization Problem


Normalizing training sets
$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$
x_2
x_1
3
5
Subtract mean:
$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$
$x := x - \mu$
Normalize variance
$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} ** 2$
element-wise
$x /= \sigma$
Use same μ σ to normalize test set.
Andrew Ng

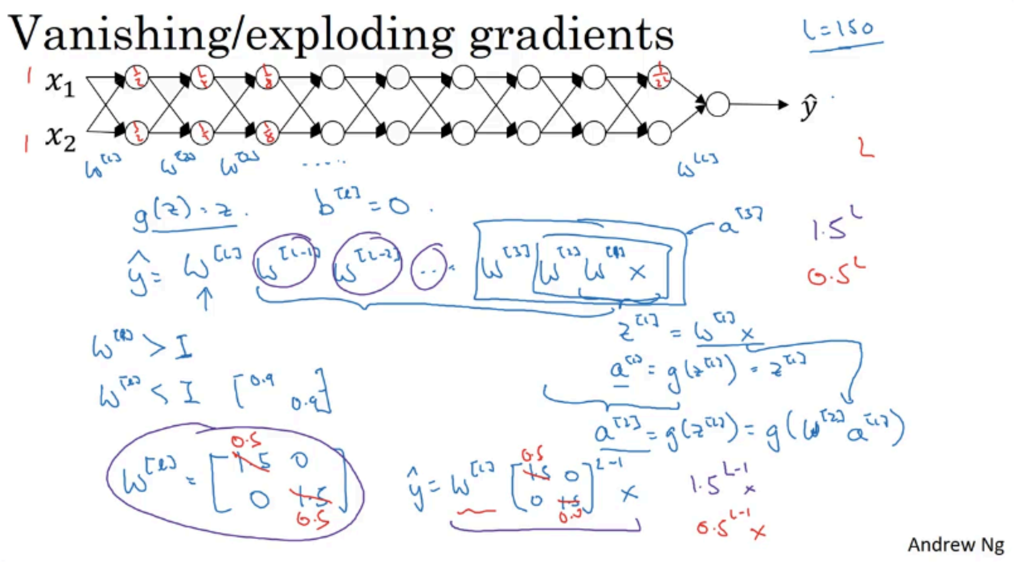
If you use this to scale your training data, then use the same mu and sigma to normalize your test set.

In particular, you don't want to normalize the training set and a test set differently. Whatever this value is and whatever this value is, use them in these two formulas so that you scale your test set in exactly the same way rather than estimating mu and sigma squared separately on your training set and test set, because you want your data both training and test examples to go through the same transformation defined by the same Mu and Sigma squared calculated on your training data. → normalize both train and test set



Cost function after taking normalize → its easy for gradient descent to work

Vanishing / Exploding Gradients


Vanishing/exploding gradients
$L = 150$
x_1
x_2
$\hat{y}$
$W^{[1]}$ $W^{[2]}$ $W^{[3]}$ ... $W^{[L]}$
$g(z) = z$
$b^{[l]} = 0$
$\hat{y} = W^{[L]} W^{[L-1]} W^{[L-2]} \cdots W^{[3]} W^{[2]} W^{[1]} x$
$a^{[2]}$
1.5^L
0.5^L
$z^{[1]} = W^{[1]} x$
$a^{[1]} = g(z^{[1]}) = z^{[1]}$
$a^{[2]} = g(z^{[2]}) = g(W^{[2]} a^{[1]})$
$W^{[l]} > I$
$W^{[l]} < I$ $\begin{bmatrix} 0.9 & \\ & 0.9 \end{bmatrix}$
$W^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$ 0.5
$\hat{y} = W^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} x$
$1.5^{L-1} x$
$0.5^{L-1} x$
Andrew Ng

So the intuition I hope you can take away from this is that at the weights W, if they're all just a little bit bigger than one or just a little bit bigger than the identity matrix, then with a very deep network the activations can explode. And if W is just a little bit less than identity. So this maybe here's 0.9, 0.9, then you have a very deep network, the activations will decrease exponentially.

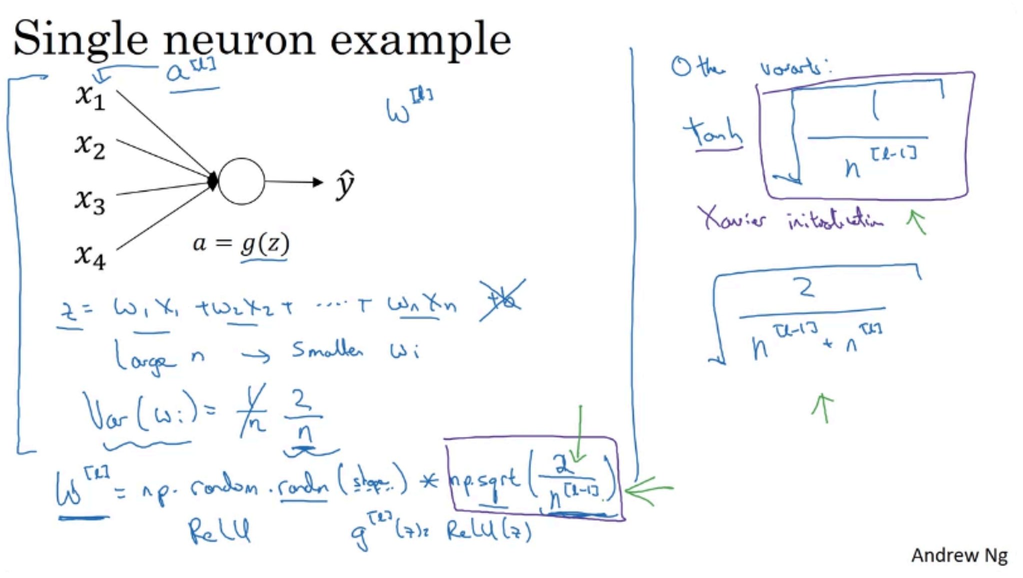
- if we choose W > I (identity matrix) → the weight could be very large (increase exponentially)
- if we choose W < I → the weight could be very small (decrease exponentially)

But with such a deep neural network, if your activations or gradients increase or decrease exponentially as a function of L, then these values could get really big or really small.

And this makes training difficult, especially if your gradients are exponentially smaller than L, then gradient descent will take tiny little steps. It will take a long time for gradient descent to learn anything.

To summarize, you've seen how deep networks suffer from the problems of vanishing or exploding gradients. In fact, for a long time this problem was a huge barrier to training deep neural networks. It turns out there's a partial solution that doesn't completely solve this problem but it helps a lot which is careful choice of how you initialize the weights.

Weight Initialization for Deep Networks

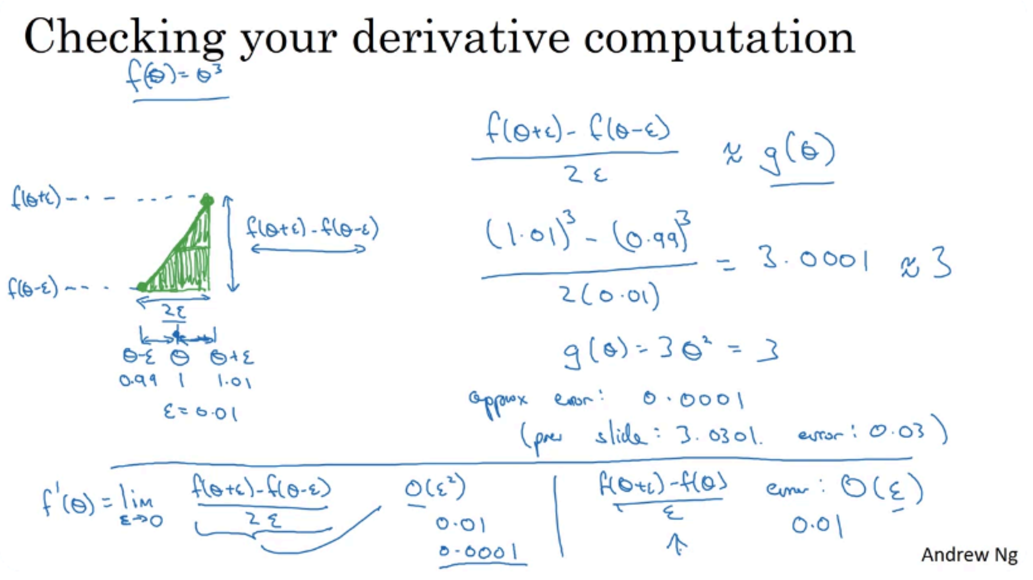

Single neuron example
$a^{[l]}$
$w^{[l]}$
x_1
x_2
x_3
x_4
$\hat{y}$
$a = g(z)$
$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n$
large n → smaller w_i
$Var(w_i) = \frac{1}{n}$ $\frac{2}{n}$
$W^{[l]}$ = np.random.randn(shape) * np.sqrt($\frac{2}{n^{[l-1]}}$)
ReLU
$g^{[l]}(z) = ReLU(z)$
Other variants:
tanh
$\sqrt{\frac{1}{n^{[l-1]}}}$
Xavier initialization
$\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$
Andrew Ng

So in order to make z not blow up and not become too small, you notice that the larger n is, the smaller you want w_i to be, right? Because z is the sum of the $w_i x_i$. And so if you're adding up a lot of these terms, you want each of these terms to be smaller. One reasonable thing to do would be to set the variance of w to be equal to $\frac{1}{n}$, where n is the number of input features that's going into a neuron.

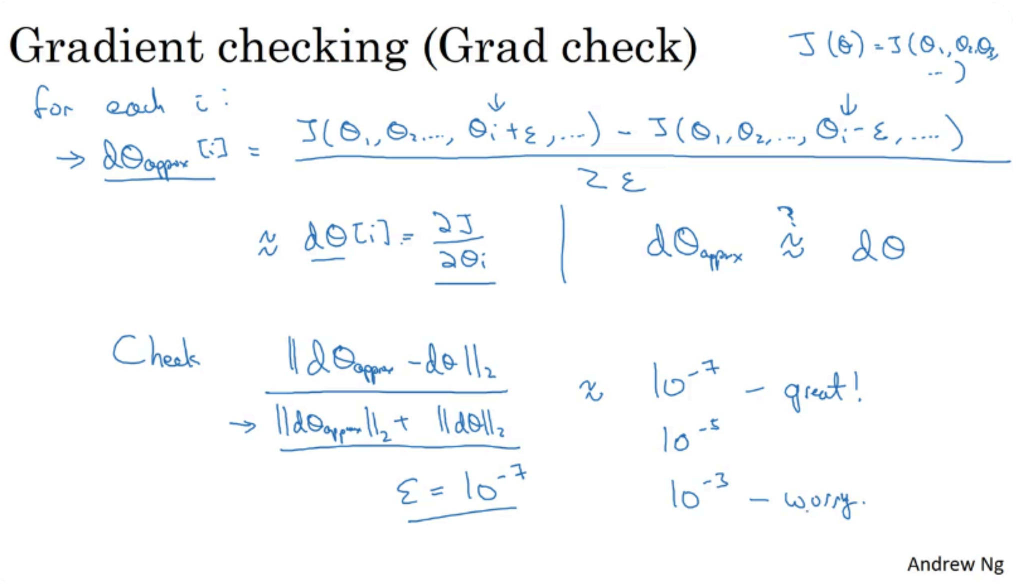
But for working a little bit better when we use:

- ReLU function, Variance of w should be $\frac{2}{n}$, then we should initialize W with np.random.rand(shape) * np.sqrt($\frac{2}{n^{[l-1]}}$)
- tanh function: instead of using $\frac{2}{n^{[l-1]}}$ we could use $\frac{1}{n^{[l-1]}}$, this is called Xavier initalization, and there is another version $\frac{2}{n^{[l-1]}+n^{[l]}}$

Numerical Approximation of Gradients


Checking your derivative computation
$f(\theta) = \theta^3$
$f(\theta+\varepsilon)$
$f(\theta-\varepsilon)$
$f(\theta+\varepsilon) - f(\theta-\varepsilon)$
2ε
$\theta-\varepsilon$ θ $\theta+\varepsilon$
0.99 1 1.01
$\varepsilon = 0.01$
$\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta)$
$\frac{(1.01)^3 - (0.99)^3}{2(0.01)} = 3.0001 \approx 3$
$g(\theta) = 3\theta^2 = 3$
approx error: 0.0001
(prev slide: 3.0301. error: 0.03)
$f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$
$O(\varepsilon^2)$
0.01
0.0001
$\frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon}$
error: $O(\varepsilon)$
0.01
Andrew Ng

Gradient Checking



Gradient checking (Grad check)

$J(\theta) = J(\theta_1, \theta_2, \theta_3, \ldots)$

for each i:

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

$$\approx d\theta[i] = \frac{\partial J}{\partial \theta_i} \qquad d\theta_{approx} \overset{?}{\approx} d\theta$$

Check $$\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2} \approx \begin{cases} 10^{-7} - \text{great!} \\ 10^{-5} \\ 10^{-3} - \text{worry.} \end{cases}$$

$\varepsilon = 10^{-7}$

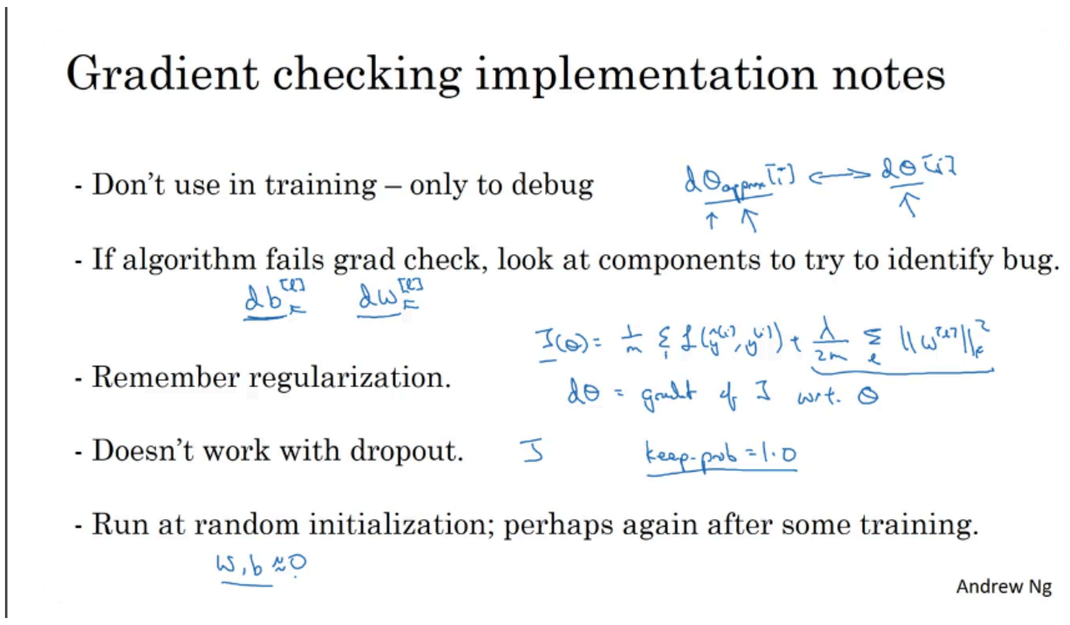
Andrew Ng

check if our gradient is right:

using formula in the image

$$\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2}$$

Gradient Checking Implementation Notes



Gradient checking implementation notes

- Don't use in training – only to debug $\quad d\theta_{approx}[i] \longleftrightarrow d\theta[i]$
- If algorithm fails grad check, look at components to try to identify bug. $\quad db^{[l]} \quad dw^{[l]}$
- Remember regularization. $\quad J(\theta) = \frac{1}{m}\sum_i \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_l \|w^{[l]}\|_F^2 \quad d\theta = \text{grad of } J \text{ wrt. } \theta$
- Doesn't work with dropout. $\quad J \quad$ keep-prob $= 1.0$
- Run at random initialization; perhaps again after some training. $\quad w, b \approx 0$

Andrew Ng

 There isn't an easy to compute cost function J that dropout is doing gradient descent on.

It turns out that dropout can be viewed as optimizing some cost function J, but it's cost function J defined by summing over all exponentially large subsets of nodes they could eliminate in any iteration. So the cost function J is very difficult to compute, and you're just sampling the cost function every time you eliminate different random subsets in those we use dropout. So it's difficult to use grad check to double check your computation with dropouts. So what I usually do is implement grad check without dropout. So if you want, you can set keep-prob and dropout to be equal to 1.0.

And then turn on dropout and hope that my implementation of dropout was correct.