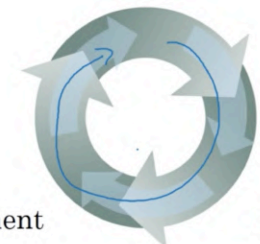
Applied ML is a highly iterative process

- # layers
- # hidden units
- learning rates
- activation functions
- ...



NLP, Vision, Speech, Structural data

Ads search Security logistic ....

Andrew Ng

Train/dev/test sets

Data | training set | dev | test

- Hold-out cross validation
- Development set "dev"

test

Previous era: 70/30% (train / test)

60/20/20%

100 - 1000 - 10000

Big data: 1,000,000 10,000 10,000

98/1/1%

99.5/ ~~.25~~ .4/ ~~.25~~ .1%

Andrew Ng

Mismatched train/test distribution

Cats

Training set:
Cat pictures from webpages

Dev/test sets:
Cat pictures from users using your app

→ Make sure dev and test come from same distribution.

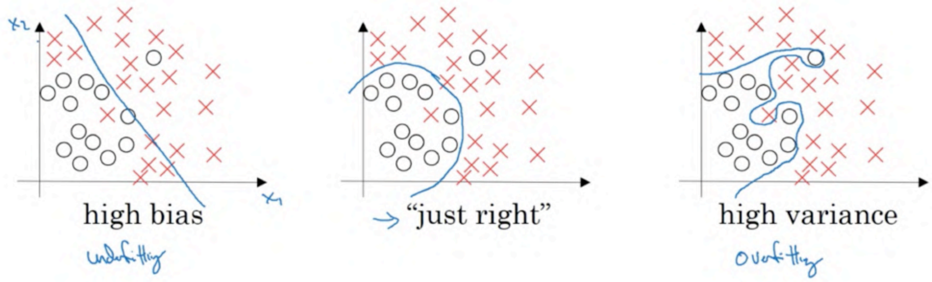
train / dev ("test")

train / test → train / dev

Not having a test set might be okay. (Only dev set.)

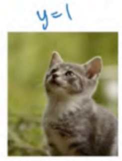
Andrew Ng

Bias and Variance



Bias and Variance

Cat classification



Variance
- The greater the difference between train and dev set error, the greater the variance

Bias
- The greater the difference between train and set dev errors from 0, the greater the bias

Doing well on train
poor on dev

Train set error:	1%	15%	15%	0.5%
Dev set error:	11%	16%	30%	1%
	high variance	high bias	high bias & high variance	low bias low variance

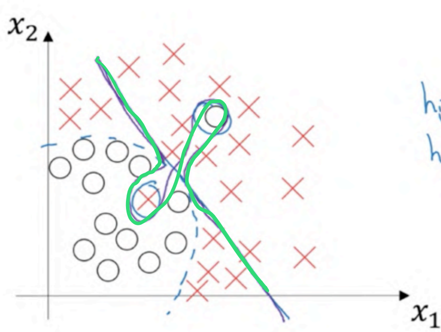
assuming → Human: ≈0%

Optimal (Bayes) error: ≈ ~~0%~~ 15%   Blurry images

baseline
can't go below

it's not even fitting the training data seam that well, then this is underfitting the data. And so this algorithm has high bias. But in contrast, this actually generalizing at a reasonable level to the dev set, whereas performance in the dev set is only 1% worse than performance in the training set. So this algorithm has a

High bias and high variance



high bias
high variance

Basic recipe for machine learning

High bias? (training data performance) — Y → Bigger network; Train longer; (NN architecture search)

↓ N

High variance? (dev set performance) — Y → More data; Regularization; (NN architecture search)

↓ N

Done

"Bias variance tradeoff"

In the earlier era of machine learning, there used to be a lot of discussion on what is called the bias variance tradeoff. And the reason for that was that, for a lot of the things you could try, you could increase bias and reduce variance, or reduce bias and increase variance. But back in the pre-deep learning era, we didn't have many tools, we didn't have as many tools that just reduce bias or that just reduce variance without hurting the other one. But in the modern deep learning, big data era, so long as you can keep training a bigger network, and so long as you can keep getting more data, which isn't always the case for either of these, but if that's the case, then getting a bigger network almost always just reduces your bias without necessarily hurting your variance, so long as you regularize appropriately. And getting more data pretty much always reduces your variance and doesn't hurt your bias much. So what's really happened is that, with these two steps, the ability to train, pick a network, or get more data, we now have tools to drive down bias and just drive down bias, or drive down variance and just drive down variance, without really hurting the other thing that much. And I think this has been one of the big reasons that deep learning has been so useful for supervised learning, that there's much less of this tradeoff where you have to carefully balance bias and variance, but sometimes you just have more options for reducing bias or reducing variance without necessarily increasing the other one.

Andrew Ng

Logistic regression

$$\min_{w,b} J(w,b)$$

$w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

λ = regularization parameter

lambda lambd

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2 \quad \cancel{+ \frac{\lambda}{2m} b^2}$$

omit

Euclidean → L_2 regularization $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

Manhatten → L_1 regularization $\frac{\lambda}{2m}\sum_{j=1}^{n_x} |w_j| = \frac{\lambda}{2m}\|w\|_1$

w will be sparse

Please note that in the next video at 5:45, the Frobenius norm formula should be the following:

$||w^{[l]}||^2 = \sum_{i=1}^{n^l} \sum_{j=1}^{n^{[l-1]}} (w_{i,j}^{[l]})^2$

Andrew Ng

The limit of summation of i should be from 1 to $n^{[l]}$,

The limit of summation of j should be from 1 to $n^{[l-1]}$.

(it's flipped in the video). The rows "i" of the matrix should be the number of neurons in the current layer $n^{[l]}$;

whereas the columns "j" of the weight matrix should equal the number of neurons in the previous layer $n^{[l-1]}$.

Neural network

$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{n} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|_F^2$$

$$\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} (w_{ij}^{[l]})^2 \qquad w: (n^{[l]}, n^{[l-1]}).$$

"Frobenius norm" $\quad \|\cdot\|_2^2 \quad \|\cdot\|_F^2$

$$dw^{[l]} = \boxed{(\text{from backprop}) + \frac{\lambda}{m} w^{[l]}} \qquad \frac{\partial J}{\partial w^{[l]}} = dw^{[l]}$$

$$\rightarrow w^{[l]} := w^{[l]} - \alpha\, dw^{[l]}$$

"L2" → "Weight decay"

$$w^{[l]} := w^{[l]} - \alpha\left[(\text{from backprop}) + \frac{\lambda}{m} w^{[l]}\right]$$

$$\boxed{\left(1 - \frac{\alpha\lambda}{m}\right) w^{[l]}} = \boxed{w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]}} - \alpha(\text{from backprop})$$

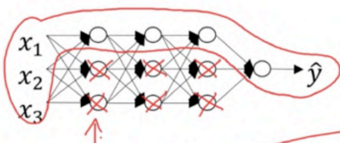
Andrew Ng

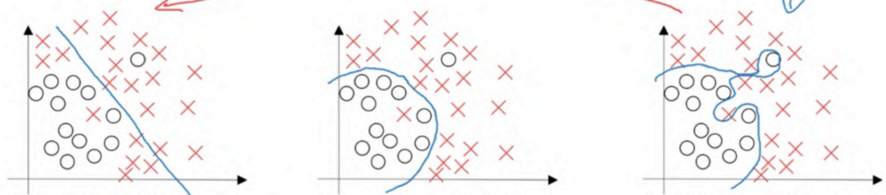
How does regularization prevent overfitting? (Variance)

$$J(w^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|_F^2$$

penalize weight matrix from being too large

$w^{[l]} \approx 0$





$\lambda\uparrow$ $W\downarrow$

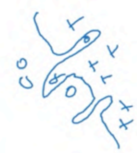
Andrew Ng

How does regularization prevent overfitting?

tanh $\quad g(z) = \tanh(z)$

W small → Z small

$\lambda\uparrow \quad w^{[l]}\downarrow \quad z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$

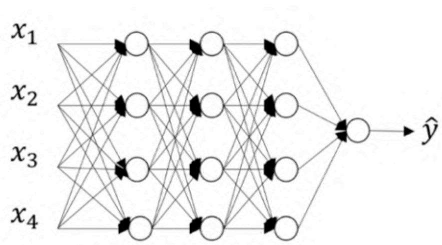


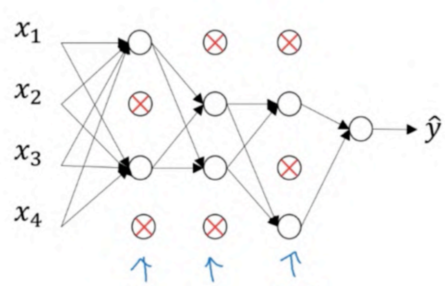
Every layer ≈ linear.

$$J(\dots) = \boxed{\sum_i \mathcal{L}(\hat{y}^{(i)}, y^{(i)})} + \frac{\lambda}{2m}\sum_l \|w^{[l]}\|_F^2$$

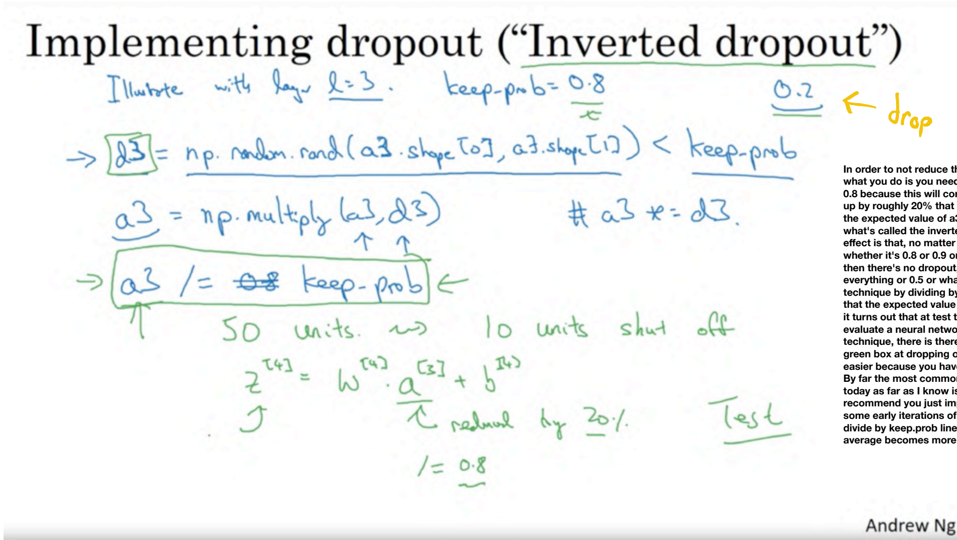
Andrew Ng

Dropout regularization

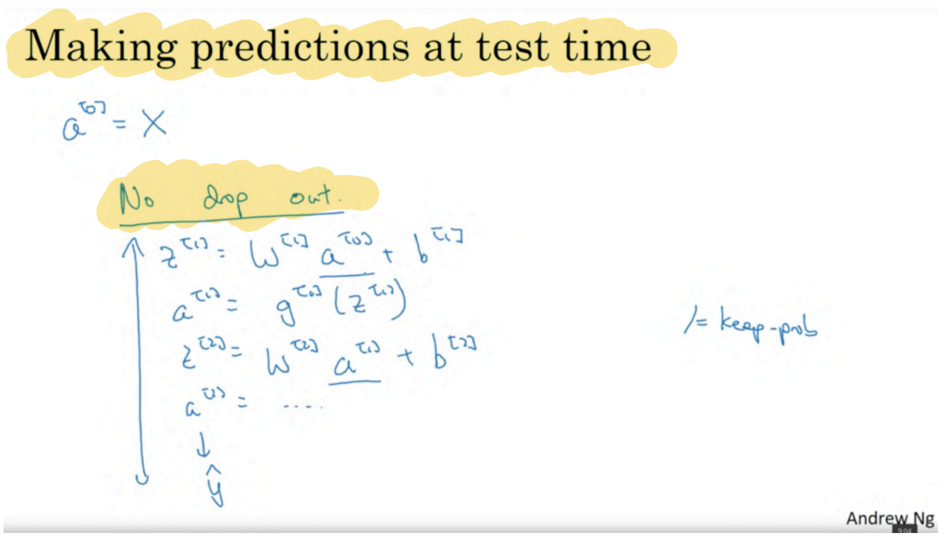


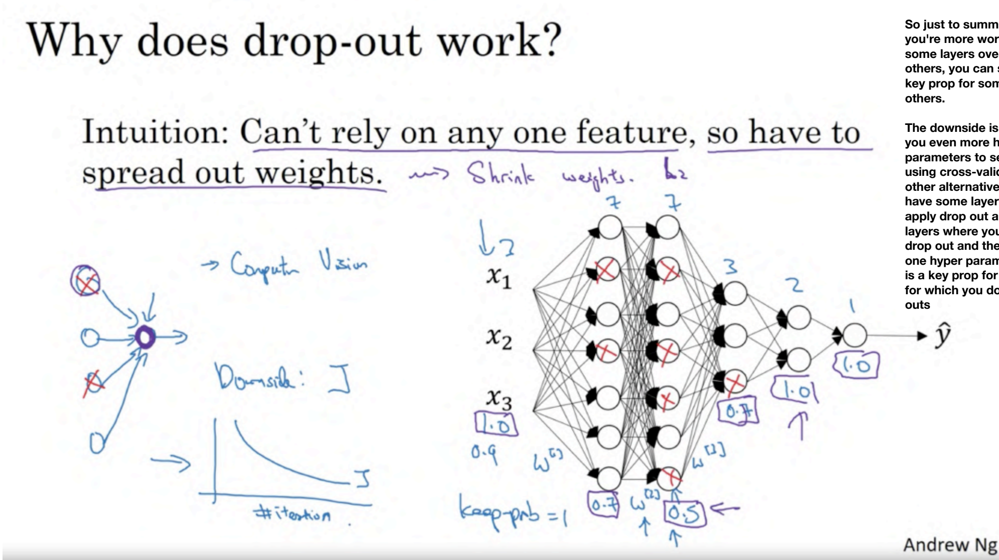


Andrew Ng



In order to not reduce the expected value of z^4, what you do is you need to take this, and divide it by 0.8 because this will correct or just a bump that back up by roughly 20% that you need. So it's not changed the expected value of a3. And, so this line here is what's called the inverted dropout technique. And its effect is that, no matter what you set to keep.prob to, whether it's 0.8 or 0.9 or even one, if it's set to one then there's no dropout, because it's keeping everything or 0.5 or whatever, this inverted dropout technique by dividing by the keep.prob, it ensures that the expected value of a3 remains the same. And it turns out that at test time, when you trying to evaluate a neural network, this inverted dropout technique, there is there is line to are due to the green box at dropping out. This makes test time easier because you have less of a scaling problem. By far the most common implementation of dropouts today as far as I know is inverted dropouts. I recommend you just implement this. But there were some early iterations of dropout that missed this divide by keep.prob line, and so at test time the average becomes more and more complicated

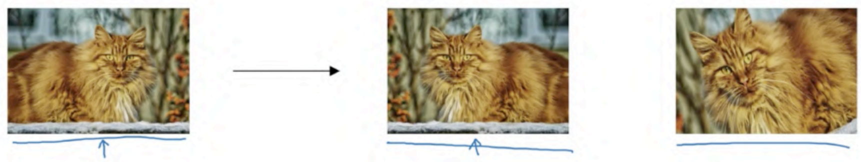




So just to summarize, if you're more worried about some layers overfitting than others, you can set a lower key prop for some layers than others.

The downside is, this gives you even more hyper parameters to search for using cross-validation. One other alternative might be to have some layers where you apply drop out and some layers where you don't apply drop out and then just have one hyper parameter, which is a key prop for the layers for which you do apply drop outs

Data augmentation



Andrew Ng

Early stopping

Can't work on either independently

Orthogonalization.

- Optimize cost function J
  - Gradient, ...
- Not overfit.
  - Regularization, ...

L_2

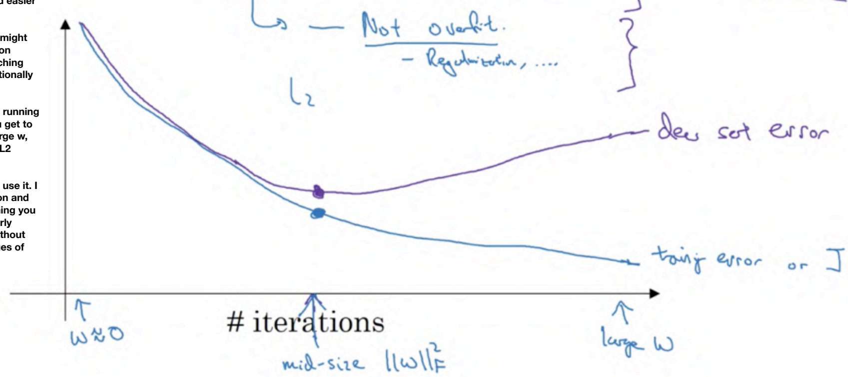
$J(w,b)$

Alternative is just use L2 regularization then you can just train the neural network as long as possible. I find that this makes the search space of hyper parameters easier to decompose, and easier to search over.

But the downside of this though is that you might have to try a lot of values of the regularization parameter lambda. And so this makes searching over many values of lambda more computationally expensive.

And the advantage of early stopping is that running the gradient descent process just once, you get to try out values of small w, mid-size w, and large w, without needing to try a lot of values of the L2 regularization hyperparameter lambda.

Despite it's disadvantages, many people do use it. I personally prefer to just use L2 regularization and try different values of lambda. That's assuming you can afford the computation to do so. But early stopping does let you get a similar effect without needing to explicitly try lots of different values of lambda.

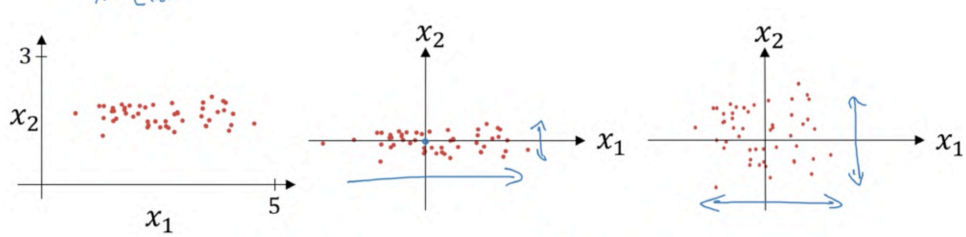


Andrew Ng

Normalizing training sets

Speeds up training

$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$



Subtract mean:

$$\boxed{\mu} = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

$$\boxed{x := x - \mu}$$

Normalize variance

$$\boxed{\sigma^2} = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} ** 2$$

← element-wise

$$\boxed{x \mathbin{/}= \sigma^2}$$

Use same μ σ^2 to normalize test set.

Andrew Ng

Why normalize inputs?

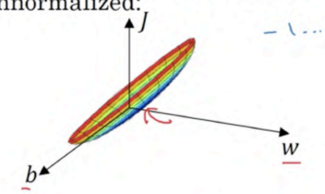
$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

w_1 x_1: $1 \ldots 1000$ ←

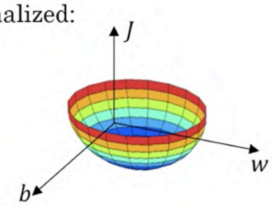
w_2 x_2: $0 \ldots 1$ ←

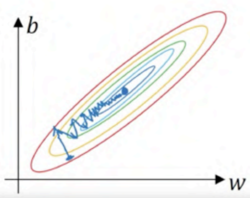
$-1 \ldots 1$

Unnormalized:



Normalized:

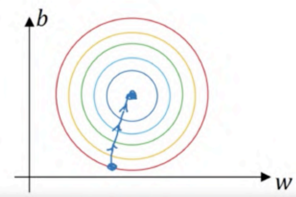




$x_1: 0 \ldots 1$

$x_2: -1 \ldots 1$

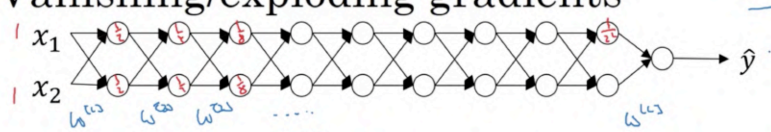
$x_3: 1 \ldots 2$



Andrew Ng

Vanishing/exploding gradients

$L = 150$



$W^{[1]}$ $W^{[2]}$ $W^{[3]}$ .... $W^{[L]}$

L

$g(z) = z$. $b^{[l]} = 0$.

$\hat{y} = W^{[L]} W^{[L-1]} W^{[L-2]} \cdots W^{[3]} W^{[2]} W^{[1]} x$ — $a^{[3]}$

1.5^L

0.5^L

$z^{[1]} = W^{[1]} x$

$a^{[1]} = g(z^{[1]}) = z^{[1]}$

$a^{[2]} = g(z^{[2]}) = g(W^{[2]} a^{[1]})$

$W^{[l]} > I$

$W^{[l]} < I$ $\begin{bmatrix} 0.9 & \\ & 0.9 \end{bmatrix}$

$W^{[l]} = \begin{bmatrix} \cancel{1.5}\ 0.5 & 0 \\ 0 & \cancel{1.5}\ 0.5 \end{bmatrix}$

$\hat{y} = W^{[L]} \begin{bmatrix} \cancel{1.5}\ 0.5 & 0 \\ 0 & \cancel{1.5}\ 0.5 \end{bmatrix}^{L-1} x$

$1.5^{L-1} x$

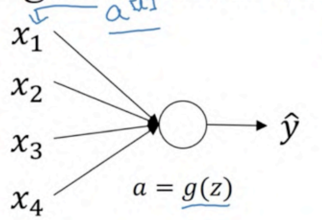
$0.5^{L-1} x$

Andrew Ng

Single neuron example

$a^{[l]}$

$W^{[l]}$



$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$ $\cancel{+b}$

Large n → Smaller w_i

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

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

ReLU $g^{[l]}(z) = \text{ReLU}(z)$

Other variants:

tanh $\boxed{\sqrt{\frac{1}{n^{[l-1]}}}}$

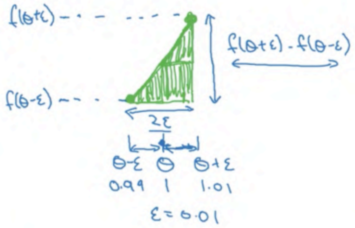
Xavier initialization

$\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$

Andrew Ng

Checking your derivative computation

$f(\theta) = \theta^3$



$$\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta)$$

$$\frac{(1.01)^3 - (0.99)^3}{2(0.01)} = 3.0001 \approx 3$$

$$g(\theta) = 3\theta^2 = 3$$

approx error: 0.0001

(prev slide: 3.0301. error: 0.03)

$$f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \qquad O(\varepsilon^2)$$

0.01

0.0001

$$\frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon} \qquad \text{error: } O(\varepsilon)$$

0.01

Andrew Ng

Gradient check for a neural network

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

concatenate

$J(W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}) = J(\theta)$

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

concatenate

Is $d\theta$ the gradient of J?

Andrew Ng

Gradient checking (Grad check)

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

for each i:

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

$$\approx d\theta[i] = \frac{\partial J}{\partial \theta_i}$$

$d\theta_{approx} \overset{?}{\approx} d\theta$

Check $$\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2} \approx \boxed{10^{-7} \text{ - great!}}$$

10^{-5}

$\varepsilon = 10^{-7}$

$\rightarrow 10^{-3}$ - worry.

Andrew Ng

Gradient checking implementation notes

- Don't use in training – only to debug

$d\theta_{approx}[i] \longleftrightarrow d\theta[i]$

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

$db^{[l]}$ $dw^{[l]}$

- Remember regularization.

$J(\theta) = \frac{1}{m}\sum_i \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \underbrace{\frac{\lambda}{2m}\sum_l \|w^{[l]}\|_F^2}$

$d\theta$ = grad of J w.r.t. θ

- Doesn't work with dropout.

J keep-prob = 1.0

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

$w, b \approx 0$