

## Train / Dev / Test Sets

Training	Hold-out Cross-Validation Development "Dev" set	Test
----------	--	------

100 - 1000 - 10000 samples: 70 / 20 / 10

1 million - 10 millions samples: 98 / 1 / 1

## Mismatched Train / Test Distribution / Sources

Training set:

Cat pictures  
from webpages

Low resolution  
satellite imagery

Dev / Test set:

Cat pictures from users  
using my app

High resolution  
satellite imagery

The Rule of Thumb: Make sure the Dev and Test come from the same distribution.

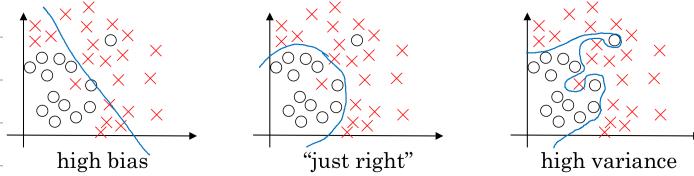
Why not also training set comes from the same distribution?

Well, it could be or it doesn't have to. I mean during the development of a model, the evaluation is done on the Dev set.

So, the model is good as long as the evaluation on Dev set is good. That's why The Rule of Thumb is talking about Dev set and Test set.

It might be OK to not have a test set. So, only train and dev set. In this case, the dev set is also called test set.

## Bias and Variance



Underfitting

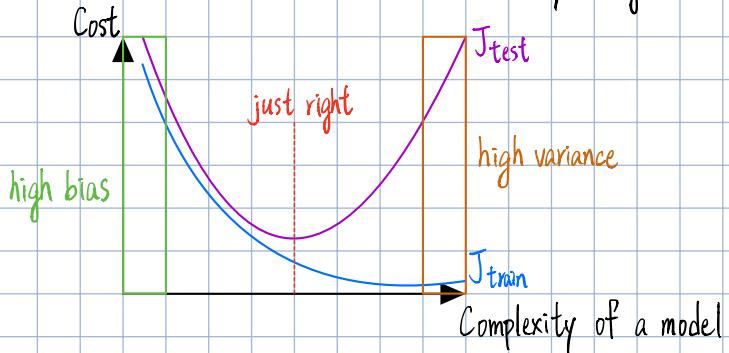
High  $J_{\text{train}} \approx J_{\text{test}}$

Overfitting

$J_{\text{test}} \gg J_{\text{train}}$   
 $J_{\text{train}}$  may be high.

What is this?  
High Bias & Variance

$J_{\text{test}} \gg J_{\text{train}}$   
 $J_{\text{train}}$  will be high.



Cat classification



Ideally, Train set error should be small and Dev set error should only be a bit larger than Train set error.

$y = 1$

$y = 0$

Cat

Non-Cat

Train set error: 1 %

15 %

15 %

0.5 %

Dev set error: 11 %

High Variance

16 %

High Bias

30 %

High Variance  
High Bias  
The Worst

1 %

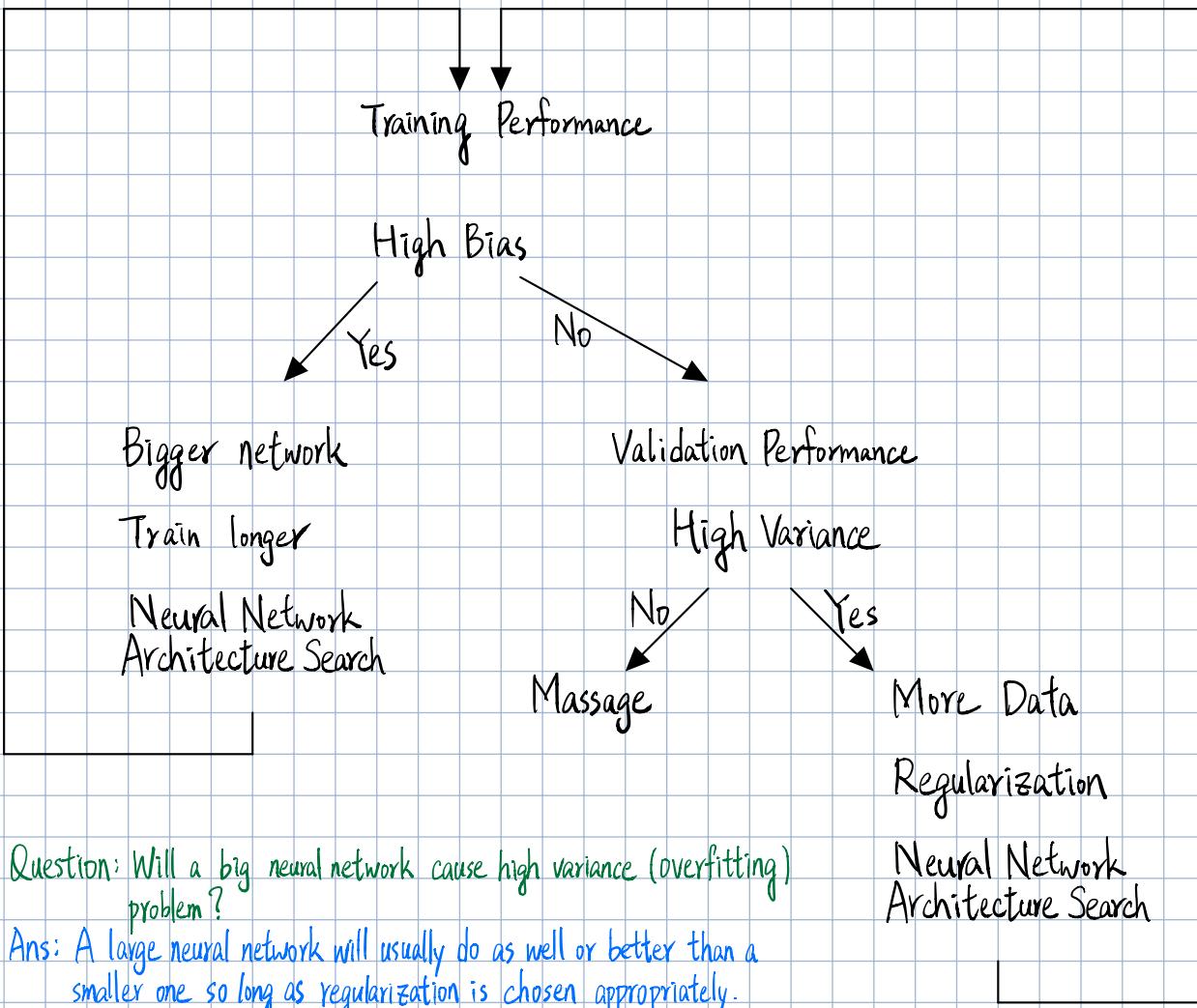
Just Right  
Low Bias & Variance

The analysis above is predicated on the assumption below:

Human/Optimal/Bayes error  $\approx 0\%$

The reason why Human error is also important is that, e.g., if human error is 15%, then this is actually pretty good.

# Basic Recipe for Machine Learning



## Bias and Variance trade-off:

Back in the pre-deep learning era, there **were** discussions about the bias and variance trade-off.

That's because we didn't have many tools that just reduce bias or just reduce variance without hurting the other one.

In the modern deep learning and big data era, as long as we can keep neural networks large and/or keep getting more data, then getting a bigger neural network almost always reduces the bias without hurting the variance and getting more data almost always reduces the variance without hurting the bias. Thus, no more trade-off between bias and variance. They can both be reduced properly.

## Regularization for Logistic Regression

Recall logistic regression:  $\min_{w,b} J(w,b) = \min_{w,b} \sum_{i=1}^m L(\hat{y}_i, y_i)$   $w \in \mathbb{R}^n$ : a vector of real numbers  
 $b \in \mathbb{R}$  : a real number

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

$$\text{L2 Regularization: } \|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$$

$$\text{L1 Regularization: } \frac{\lambda}{2m} \sum_{j=1}^{n_x} |w_j| = \frac{\lambda}{2m} \|w\|_1$$

$w$  will become sparse (vector/matrix with lots of 0)  
if L1 is used. Why?

Omit. Why only regularize weight only?

$w$  is a high dimensional vector, while  
 $b$  is just a real number. Thus, the  
majority of information is in  $w$  but  
not  $b$ . Hence, it doesn't make much  
difference.

## $\lambda$ : Regularization Parameter

And this is how regularization is done on Logistic Regression.

## Regularization for Neural Network

$$J(w^{(0)}, b^{(0)}, \dots, w^{(L)}, b^{(L)}) = \frac{1}{m} \sum_{i=1}^m L(\hat{y}_i^{(i)}, y_i^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^L \|w^{(l)}\|_F^2$$

$$\|w^{(l)}\|_F^2 = \sum_{i=1}^{n^{(l)}} \sum_{j=1}^{n^{(l-1)}} (w_{i,j}^{(l)})^2, w: (n^{(l)}, n^{(l-1)}). \text{ It's called Frobenius Norm}$$

Now, the question is how to implement gradient descent with this?

Recall before adding Regularization

$$\frac{\partial J}{\partial w^{(l)}} = dw^{(l)}: \text{ (from backpropagation)}$$

$$w^{(l)} = w^{(l)} - \alpha dw^{(l)}$$

Add Regularization

$$\frac{\partial J}{\partial w^{(l)}} = dw^{(l)}: \text{ (from backpropagation)} + \boxed{\frac{\lambda}{m} w^{(l)}}$$

$$w^{(l)} = w^{(l)} - \alpha dw^{(l)}$$

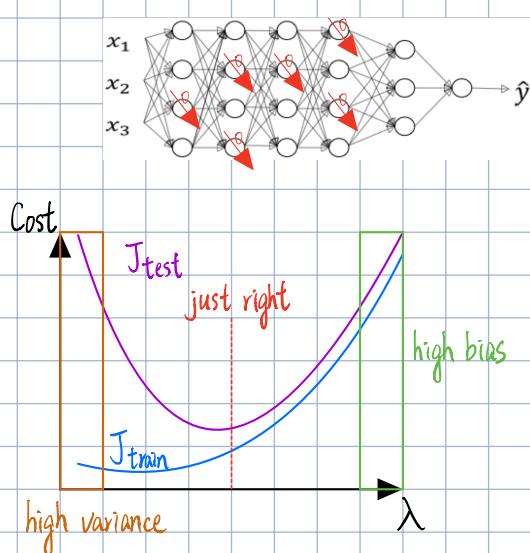
$$\rightarrow w^{(l)} = w^{(l)} - \boxed{\frac{\alpha \lambda}{m} w^{(l)}} - \alpha \text{ (from backpropagation)}$$

$$(1 - \frac{\alpha \lambda}{m}) w^{(l)}$$

$(1 - \frac{\alpha \lambda}{m})$  will always be smaller than 1, which is why L2 Regularization is also referred to as Weight Decay.

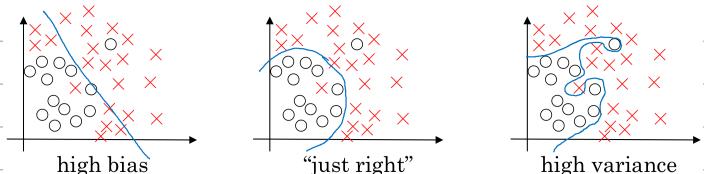
# Why Regularization Reduces Overfitting?

The first intuitive explanation:



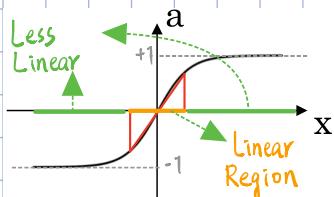
$$J(w^{(0)}, b^{(0)}, \dots, w^{(L)}, b^{(L)}) = \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^L \|w^{(l)}\|_F^2$$

One intuition is that if  $\lambda$  is very large, it will incentivize to set weights to be reasonably close to 0. If many weights for hidden units are close to 0, then it is zeroing out the impact of the hidden units. Thus, a complex neural network becomes a much simpler network. Thus, it will make it from overfitting toward underfitting. And hopefully, there will be an intermediate value of  $\lambda$  that results in the just-right case.



The second intuitive explanation:

The activation function  $g(z) = \tanh(z)$



$$w^{(0)} = w^{(0)} - \frac{\alpha \lambda}{m} w^{(0)} - \alpha \quad (\text{from backpropagation})$$

$$(1 - \frac{\alpha \lambda}{m}) w^{(0)}$$

$$\lambda \nearrow, w^{(0)} \searrow, z^{(0)} = w^{(0)} a^{(0-1)} + b^{(0)} \searrow$$

$g(z)$  will become roughly linear as if it's just linear regression.

Thus, it will use linear region in  $\tanh(z)$ .

If every layer is a linear network, then the whole network is just a linear network, which means it can't learn the nonlinearity of data.

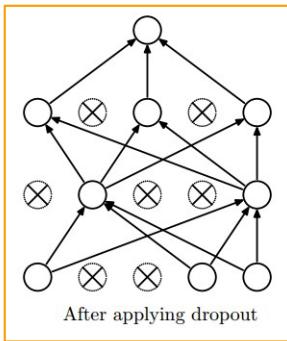
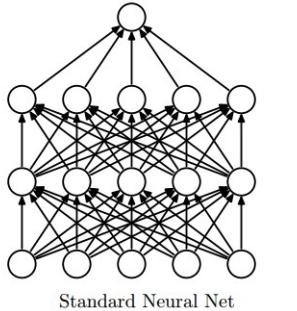
$$\begin{aligned} \hat{y} &= a^{(2)} = z^{(2)} = W^{(2)}(W^{(1)}X + b^{(1)}) + b^{(2)} \\ &= (W^{(2)}W^{(1)})X + (b^{(2)}b^{(1)}) \\ &= W'X + b' \end{aligned}$$

Thus, overfitting is reduced.

This shows that the neural network is generating a linear function of an input, which is equal to no hidden layers at all.

# Dropout Regularization

If dropout rate is 0.5, it means for each node in hidden layers, there is 50% probability that a node will be eliminated.



~~iteration of gradient descent~~  
For each ~~training sample~~, it runs through the same network but with different nodes being dropout.  
The network becomes simpler. Thus, overfitting is reduced.

Implementing Dropout ("Inverted Dropout") in Training time:

Hyperparameter:  $\lambda = 3$ , keep-prob = 0.8 (dropout rate = 0.2)

For each ~~iteration of gradient descent~~ (all samples)

$d_3 = \text{np.random.rand}(a_3.\text{shape}[0], a_3.\text{shape}[1]) < \text{keep-prob}$

$\text{np.multiply}(a_3, d_3, \text{out}=a_3) \rightarrow$  I learn it today. It's efficient code.

$a_3 /= \text{keep-prob}$

Why? Assume this layer has 50 neurons. Dropout will reset 10 units to 0.

For the next layer,  $Z^{(3)} = W^{(3)} \cdot a^{(3)} + b^{(3)}$ .

Now,  $a^{(3)}$  is reduced by 20%. In order to not reduce the expected value of  $Z^{(3)}$ ,  $a^{(3)}$  is divided by 0.8 because it will bump  $a^{(3)}$  back by 20%, which ensures the expected value of  $a^{(3)}$  is unchanged.

It's also called Inverted Dropout Technique.

Making Predictions at Testing Time: Dropout is disabled. Every units count.

$$\begin{aligned} a^{(0)} &= X \\ Z^{(1)} &= W^{(1)} a^{(0)} + b^{(1)} \\ a^{(1)} &= g^{(1)}(Z^{(1)}) \\ &\vdots \\ \hat{y} & \end{aligned}$$

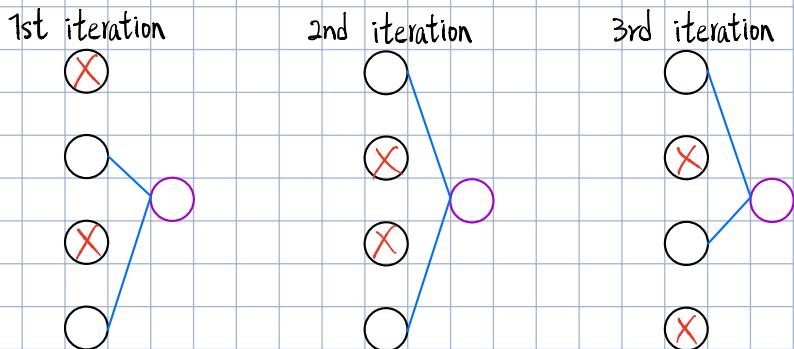
Recall that  $a_3 /= \text{keep-prob}$  is executed in training.

In testing, we don't need to implement any value

rescaling method because dropout is disabled.

# Understanding Dropout

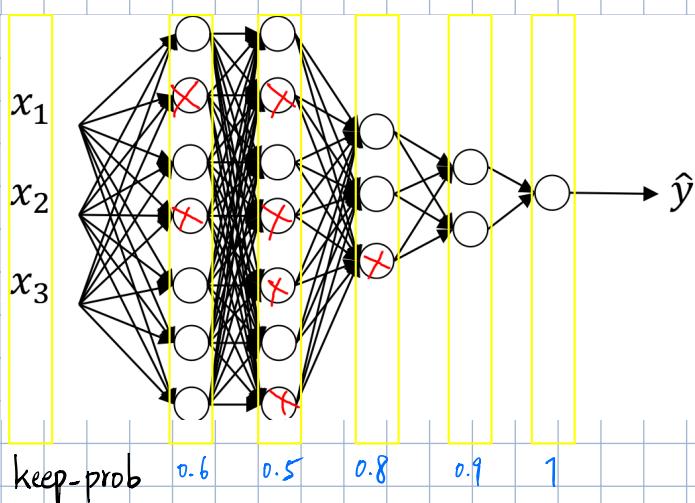
Intuition: Can't rely on any one feature, so have to spread out weights.



With dropout, the inputs can get randomly eliminated. E.g., as shown above, different units are eliminated in each iteration. So, what this means is that the unit ○ can't rely on any one feature, because any one feature could go away at random. Thus, in particular, it will be reluctant to put all of its bets on a specific input. Hence, the unit ○ will be more motivated to spread out the weights to four of its input units.

By spreading out the weights, it will tend to have an effect of shrinking the squared norm of the weights. The effect of implementing dropout is that it shrinks the weights and is similar to L<sub>2</sub> Regularization. Only that L<sub>2</sub> Regularization can be even more adaptive to the scale of different inputs.

Another important detail of dropout: keep-prob can be varied by each layer.



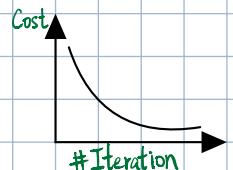
In general, the layer with more hidden units can have lower keep-prob, because more hidden units often result to overfitting.

Technically, dropout can also be applied to the input layer. But usually people don't do this. If you really want to apply dropout to the input layer, the keep-prob should be close to 1.

In Computer Vision, the feature dimension is so large such that the training data is never enough, which almost always leads to overfitting. So, dropout is frequently used in Computer Vision. However, this idea doesn't always generalize to other disciplines.

## Drawback/Downside of Dropout:

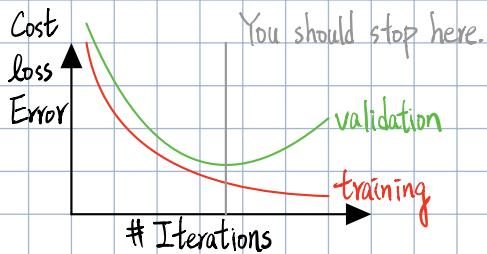
Applying dropout in training will make a neural network with units being shut off randomly. Thus, the cost may not decrease monotonically. The solution is to turn off dropout when calculating cost.



## Other Regularization Methods

- Data Augmentation:

- Early Stopping:



The downside of early stopping:

Machine Learning process can be seen as two tasks:

- Optimize cost function  $J$ : Gradient Descent, Momentum, RMS Prop, AdamW, etc.
- Avoid overfitting: More data, Regularization, etc.

To be honest, the above two points are kind of contradicting. If I want to make cost function  $J$  as small as possible, then it's likely overfitting, and vice versa.

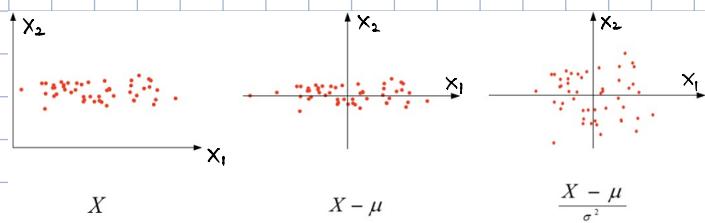
And we want to approach these two tasks independently. It's called **Orthogonalization**.

Early stopping couples these two tasks, we no longer can work on these two tasks independently, because by stopping gradient descent early, I am breaking the process of optimizing cost function  $J$ .

Andrew prefers to just use **L<sub>2</sub> Regularization** and try different values of the parameter  $\lambda$ , so that he can just train the network as long as possible. But it's more computationally expensive.

## Normalizing Inputs

Normalizing training sets.  $X \in \mathbb{R}^2$ :  $x^{(i)}, i=1, \dots, m$



Note: If the mean and the variance derived from the training set is used to normalize the training data, then the same mean and variance must be used to normalize the test set.

In particular, you don't want to normalize the training set and test set differently. We want the training set and test set to go through the same transformation defined by the same  $\mu$  and  $\sigma^2$  calculated from the training data. Intuitively, it makes sense. But, ...

OK. I don't think the normalization described above can be applied to images. If the training data is image, it doesn't make any sense to normalize this way.

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

Assume the feature  $X_1$ 's value ranges from 1 to 1000 and  $X_2$ 's value ranges from 0 to 1. It needs normalization.

If unnormalized, then the small learning rate might be used, because it might need a lot of steps to oscillate back and forth to find its way to the minimum.

If normalized, gradient descent can go straight to the minimum.

1. Subtract mean: Assume  $x^{(i)} \in \mathbb{R}^n$

$$\text{mean } \mu = \frac{1}{m} \sum_{i=1}^m x^{(i)} \quad \mu \in \mathbb{R}^n : (n, 1)$$

2. Normalize Variances:

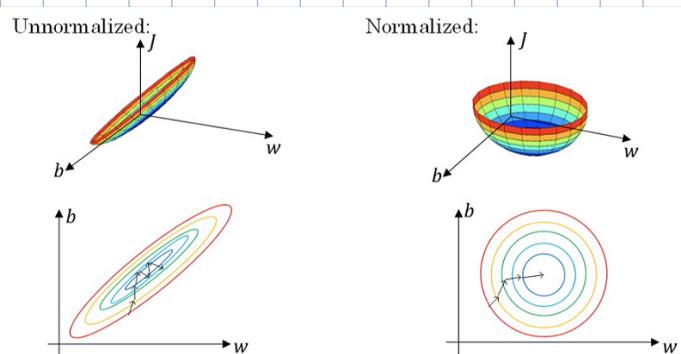
$$\text{Variance: } \sigma^2 = \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \mu)^2 \quad \sigma^2 \in \mathbb{R}^n : (n, 1)$$

$$\therefore X = \frac{X - \mu}{\sigma} \quad X : (n, m)$$

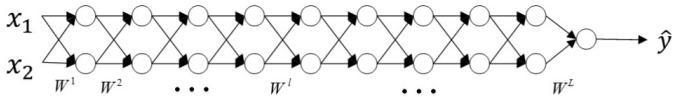
$\rightarrow$  It must be element-wise division.

$X$  will have mean 0, variance 1. But the range is not necessarily from -1 to 1.  $X$  could still be  $> 1$  and  $< -1$  after normalization.

However, if  $X_1$  ranges from 0 to 1,  $X_2$  ranges from -1 to 1 and  $X_3$  ranges from 1 to 2, these features have similar ranges, they don't need to be normalized at all. Neural networks will work just fine.



# Vanishing / Exploding Gradients



Given a very deep neural network, for the sake of simplicity, there are only two features, and  $g(z) = z$ ,  $b^{(l)} = 0$ ,

Then,  $\hat{y} = W^{[L]} W^{[L-1]} \cdots W^{[2]} W^{[1]} X$

$$z^{[l]} = z^{[1]} = \alpha^{[l]} = g^{[l]}(z^{[1]})$$

For  $l = 1, \dots, L-1$ : Case 1

Assume  $W^{[l]} = \begin{pmatrix} 1.5 & 0 \\ 0 & 1.5 \end{pmatrix}$ ,  $\hat{y} = W^{[L]} \begin{pmatrix} 1.5 & 0 \\ 0 & 1.5 \end{pmatrix}^{L-1} X$ , If  $L$  is large,  $\hat{y}$  will be exploding.

Case 2

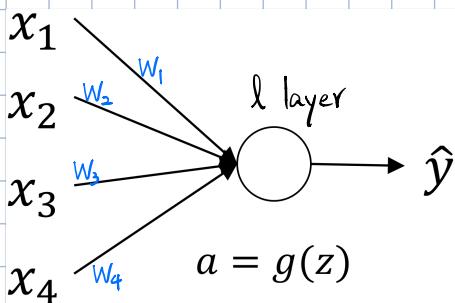
Assume  $W^{[l]} = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix}$ ,  $\hat{y} = W^{[L]} \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix}^{L-1} X$ , If  $L$  is large,  $\hat{y}$  will be vanishing.

The idea is that with a very deep neural network, the activation will increase or decrease exponentially. The similar argument can be used to show the gradient in the gradient descent is going to increase or decrease exponentially as a function of the number of layers.

Weight Initialization for Deep Networks.

*It can only partially solve the problem of exploding/vanishing gradient, not entirely.*

Single Neuron Example:



$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n \quad \text{X}$$

In order to not blow up  $z$ , the larger  $n$  is, the smaller we want  $w_1, \dots, w_n$  to be, because  $z = \sum_{i=1}^n w_i x_i$ .

Thus, we want  $\text{var}(w) = \frac{1}{n}$ ,  $n$  is the number of inputs for a neuron.

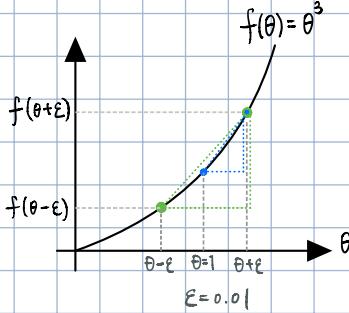
$$w^{[l]} = \underbrace{\text{np.random.rand(shape)}}_{\substack{\text{the shape of weight} \\ \text{for } l \text{ layer}}} * \underbrace{\text{np.sqrt}(\frac{1}{n^{(l-1)}})}_{\substack{\text{the number of neurons going} \\ \text{into a neuron in } l \text{ layer}}}$$

It turns out that if  $g(z) = \text{ReLU}(z)$ , then it would be better to use  $\text{var}(w) = \frac{2}{n}$ . Thus,  $\text{np.sqrt}(\frac{2}{n^{(l-1)}})$ .

Andrew mentions if  $g(z) = \tanh(z)$ , then  $\sqrt{\frac{1}{n^{(l-1)}}}$  or  $\sqrt{\frac{2}{n^{(l-1)} + n^{[l]}}}$  can be used to replace  $\text{np.sqrt}(\frac{2}{n^{(l-1)}})$ .

Xavier Initialization

# Numerical Approximation of Gradients



Goal: Find slope of  $\theta$ .  $f'(\theta) = g(\theta) = 3\theta^2 = 3$  — the ground truth

It turns out using the green triangle is better than using the blue triangle

$$\frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon} \approx g(\theta) \rightarrow \text{gradient} \Rightarrow \frac{1.01^3 - 0.99^3}{2 \times 0.01} = 3.001 | \frac{3.001 - 3}{3.001 - 3} = 0.001$$

$$\frac{f(\theta + \epsilon) - f(\theta)}{\epsilon} = \frac{1.01^3 - 1^3}{0.01} = 3.030 | \frac{3.030 - 3}{3.030 - 3} = 0.030$$

Gradient Checking: It's a technique to help finding bugs in the implementation of gradient descent.

Step 1: Concatenate  $w^{(1)}, b^{(1)}, \dots, w^{(L)}, b^{(L)}$  and reshape into a big vector  $\theta$ .

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

Step 2: Concatenate  $dw^{(1)}, db^{(1)}, \dots, dw^{(L)}, db^{(L)}$  and reshape into a big vector  $d\theta$ .

Now the big question is: Is  $d\theta$  the gradient of the cost function  $J$ ?

Step 3:  $J(\theta) = J(\theta_1, \theta_2, \theta_3, \dots)$

for each  $i$ :

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

$$\text{Ideally, } d\theta_{\text{approx}}[i] \approx d\theta[i] = \frac{\partial J}{\partial \theta_i}$$

After calculating  $d\theta_{\text{approx}}[i]$  for each  $i$ , now we have  $d\theta_{\text{approx}}$ .

Then we can calculate how close  $d\theta_{\text{approx}}$  and  $d\theta$  is.

$$\text{Euclidean distance } \text{dist}(d\theta_{\text{approx}}, d\theta) = \frac{\|d\theta_{\text{approx}} - d\theta\|_2}{\|d\theta_{\text{approx}}\|_2 + \|d\theta\|_2}$$

Andrew selects  $\epsilon = 10^{-7}$ , if  $\text{dist}(d\theta_{\text{approx}}, d\theta)$  is  $10^{-7}$  great

$10^{-5}$  something might go wrong

$10^{-3}$  something is definitely wrong

## Gradient Checking Implementation Notes

- Don't use in Training. Only to debug.
- If algorithm fails gradient checking, check components to try to identify bugs.
- Remember Regularization.
- Don't work with dropout.
- Run at random initialization; perhaps again after some training.