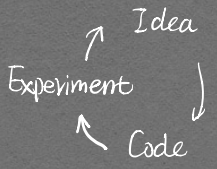
Train / Dev / Test sets

Applied ML is a highly iterative process.



Intuitions are not easily transferrable from NLP to Speech Recognition / CV.

Train / Dev / Test 100 - 1000 - 10000

70% 30%

60% 30% 10%

Train / Dev / Test 1000000

98% 1% 1%

99.5% .4% .1%

Mismatched Train/Test Distribution

Make sure Dev and Test sets come from the same distribution

Not having a Test set might be okay (Only Dev set) if we don't need an unbiased estimate of model accuracy.

Bias / Variance

	Case 1	Case 2	Case 3	Case 4
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
Human Error	.5%	1%		
(Optimal) Bayes Error				

Basic "Recipe" for Machine Learning

High Bias? (Traing Data Performance) — Yes → Bigger Network, Train Longer, NN Architecture

↓ No

High Variance? (Dev Set Performance) — Yes → More Data, Regularization, NN Architecture

↓ No

Done

"Cancels" Bias/Variance Tradeoff partially

Regularizing Your Network

Logistic Regression

$$\min_{w,b} J(w,b) \quad w \in \mathbb{R}^{n_x}, \; b \in \mathbb{R}$$

$$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \begin{cases} \boxed{\frac{\lambda}{2m} \|w\|^2} \\ \boxed{\frac{\lambda}{2m} \|w\|_1} \end{cases}$$

L_2 Regularization

$$\|w\|^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$$

L_1 Regularization

$$\|w\|_1 = \sum_{j=1}^{n_x} |w_j|$$

w will be sparse

Neural Network

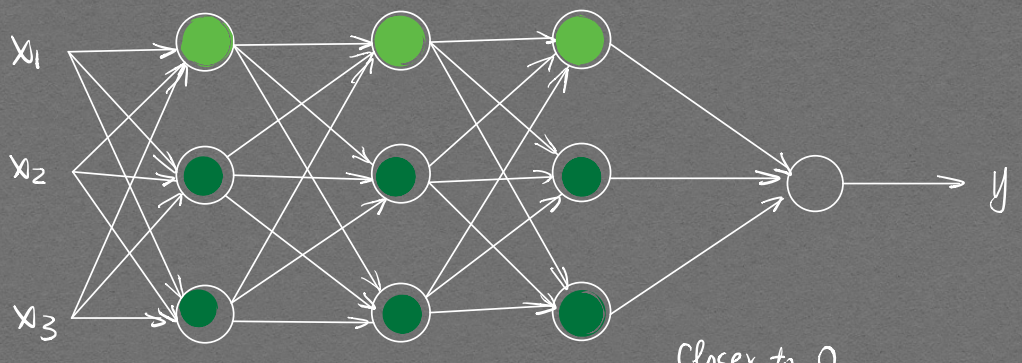
$$J(w^{[1]}, b^{[1]}, \cdots, w^{[\ell]}, b^{[\ell]}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{\ell=1}^{L} \|w^{[\ell]}\|^2$$

# of neurons in layer ℓ → # neurons in the previous layer $\ell-1$

Frobenius Norm: $\|w^{[\ell]}\|_F^2 = \sum_{i=1}^{n^{\ell}} \sum_{j=1}^{n^{[\ell-1]}} (w_{i,j}^{[\ell]})^2$ $\quad w^{[\ell]} \sim n^{[\ell]} \times n^{[\ell-1]}$

Calculate Gradient : $dw^{[\ell]} = (\text{from backprop}) + \frac{\lambda}{m} w^{[\ell]}$

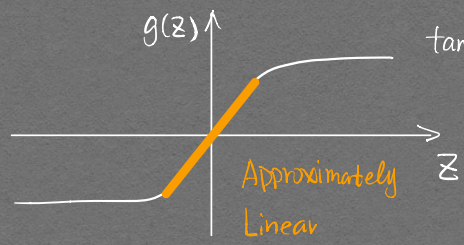
Update Step : $w^{[\ell]} = w^{[\ell]} - \alpha \, dw^{[\ell]}$ "Weight Decay"



Closer to 0

$\lambda \longrightarrow$ Inf

$w \longrightarrow 0$ $\quad z^{[\ell]} = w^{[\ell]} a^{[\ell-1]} + b^{[\ell]} \longrightarrow 0$

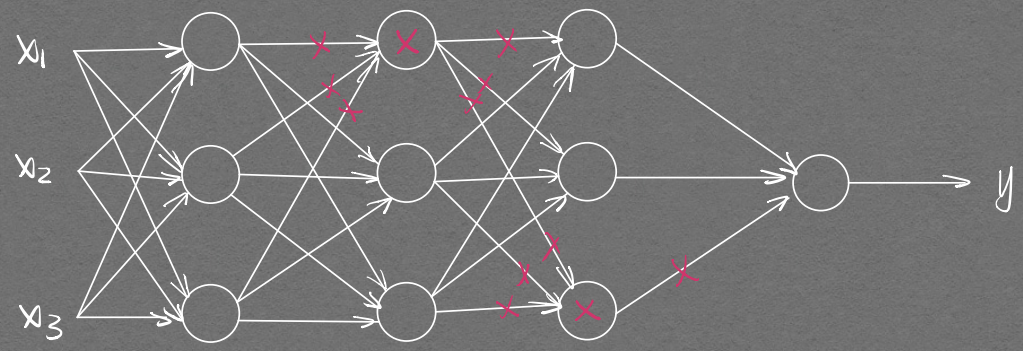


$z^{[l]}$ in the Linear Zone

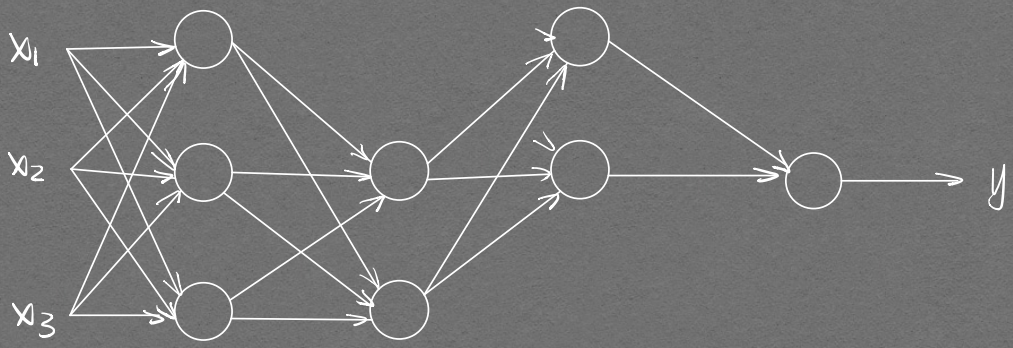
Activation function becomes more linear

Neural Network becomes more linear

Dropout Regularization



kill neurons and their input / output edges



Training Time (Inverted Dropout) : keep_prob = .8

At layer 3 :

d_3 = np.random.rand (a_3.shape[0], a_3.shape[1])

< keep_prob A True, False Mask

a_3 = np.multiply (a_3, d_3) Element-wise Multiplication

a_3 /= keep_prob Inverted Dropout

to keep expected value of a_3 unchanged

Test Time: No Dropout

(We don't want random outputs)

Understanding Dropout

1) Smaller Network

2) Input Features are killed randomly, thus weights are encouraged to Spread out

3) Different Dropout / keep-prob Ratio in Different layers

Other Regularization Methods

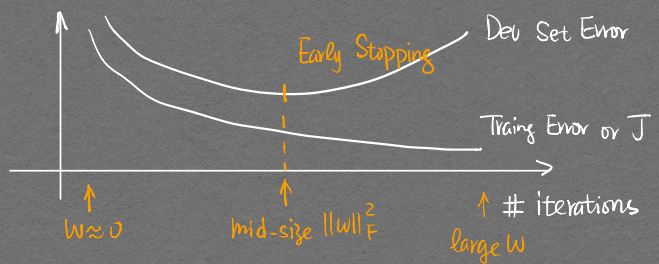
Data Augmentation (Mirroring Images, Random Cropping Images, ...)

Early Stopping

↳ Optimize J

↳ Not overfitting

Affecting Both

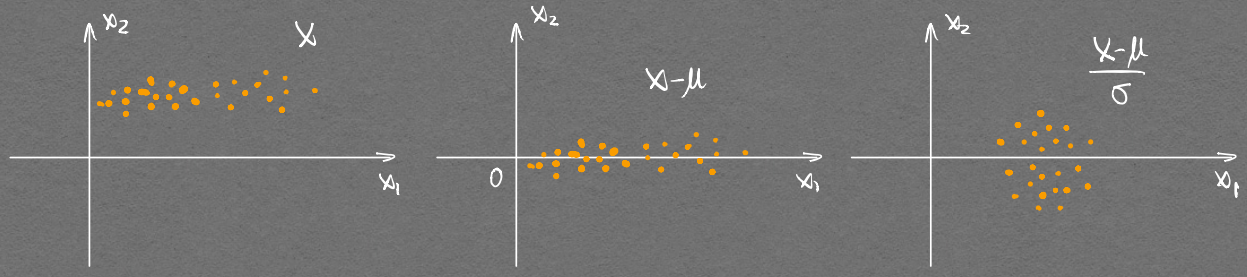


Setting Up Your Optimization Problem

Normalizing Inputs

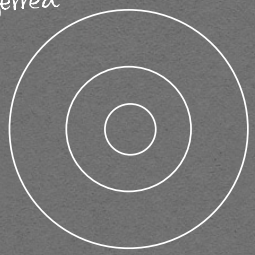
Calculate μ, σ with training set.

Use same μ, σ for test set.



Avoid Narrow Valley

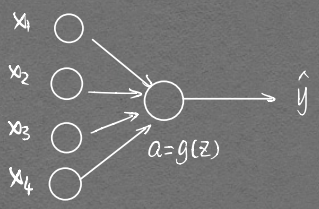
Round Shape is preferred



Vanishing / Exploding Gradients

→ Partial Solution: Weight Initialization

Single Neuron Example



$b=0$

$z = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4$

large n → small w_i

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

Other variants $w^{[l]} = np.random.randn(shape) * np.sqrt(c)$

If tanh()

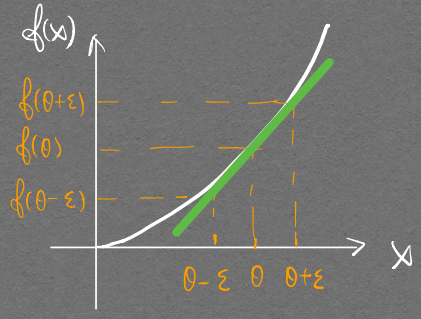
use Xavier activation, $c = \sqrt{\frac{1}{n^{[l-1]}}}$

If relu()

use He activation, $c = \sqrt{\frac{2}{n^{[l-1]}}}$

Numerical Approximation of Gradients

Checking your derivative computation



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

gradient at θ

error in the order of $O(\varepsilon^2)$

Gradient Checking

$\theta = (w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \cdots, w^{[l]}, b^{[l]})$

$d\theta = (dw^{[1]}, db^{[1]}, dw^{[2]}, db^{[2]}, \cdots, dw^{[l]}, db^{[l]})$

for each i in $1 : len(\theta)$:

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

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

pick $\varepsilon = 10^{-7}$

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

10^{-3} - wrong

Implement Gradient Checking

Don't use grad check in training - only to debug

If fails, look at components

Remember regularization term in the cost function when calculating gradients

Doesn't work with dropout. (keep-prob = 1, pass grad check, turn on dropout)

Run at random initialization, perhaps after some training.

Notes from Jupyter Notebook

Zero Initialization → Fail to break symmetry

Every neuron in each layer will learn the same thing.

We end up training a neural network with $n^{[l]} = 1$ for every layer. The network is a lot less powerful.

Random Initialization → Initialize with very large values

→ Vanishing / Exploding gradients

Slows optimization algorithm

He Initialization → Works well for Relu activation