DEEP LEARNING

Mathematical objects in Linear Algebra

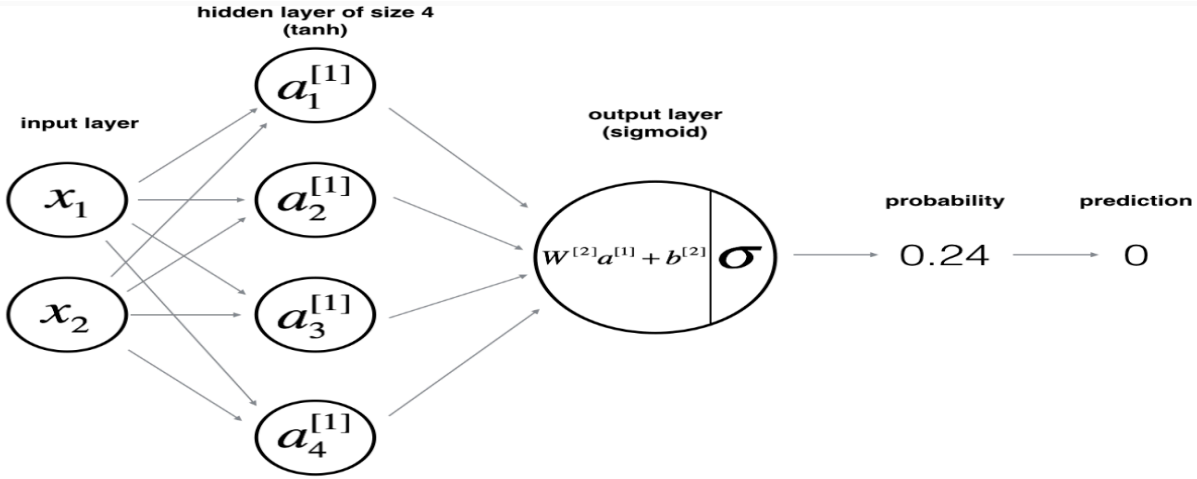
- Scalar: A scalar is just a single number
- Vectors: A vector is an array of numbers. The numbers are arranged in order.
- Matrices: A matrix is a 2-D array of numbers
- Tensor: An array of numbers arranged on variable number of axes

Deep Feedforward Networks

- Information flows through the function being evaluated from x
- Outputs of model are not fed back into itself (such models are called recurrent neural network)
- Applications: CNN used for object recognition from photos, NLP
- Represented by composing many different functions and directed acyclic graph

$$f(x) = f(3)(f(2)(f(1)(x)))$$

Deep FeedForward Networks



Mathematically:

For one example $x^{(i)}$:

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]} \tag{1}$$
$$a^{[1](i)} = \tanh(z^{[1](i)}) \tag{2}$$
$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]} \tag{3}$$
$$\hat{y}^{(i)} = a^{[2](i)} = \sigma(z^{[2](i)}) \tag{4}$$
$$y^{(i)}_{prediction} = \begin{cases} 1 & \text{if } a^{[2](i)} > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

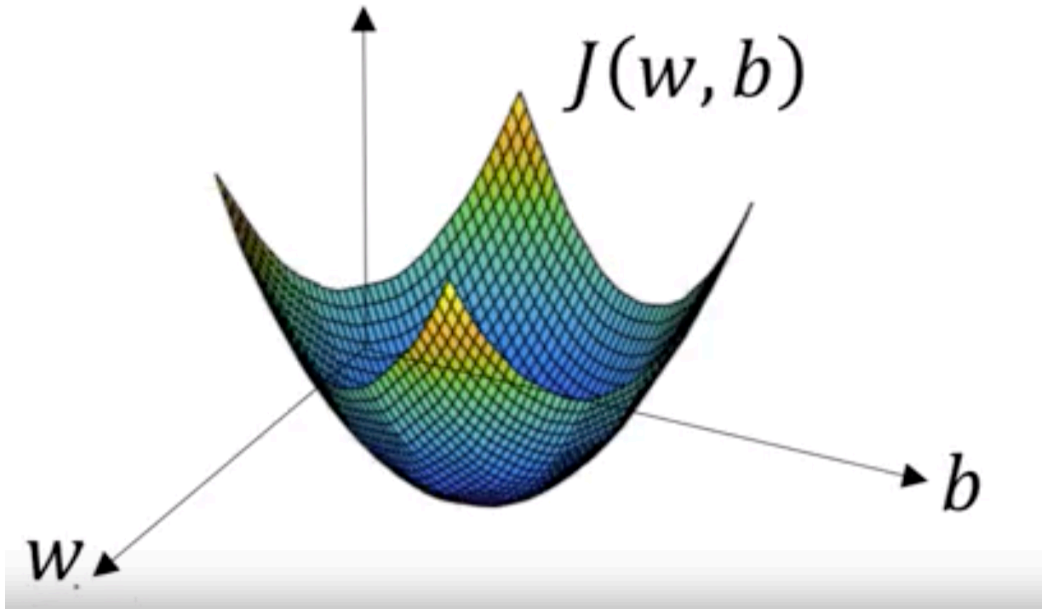
Given the predictions on all the examples, you can also compute the cost J as follows:

$$J = -\frac{1}{m} \sum_{i=0}^{m} \left(y^{(i)} \log\left(a^{[2](i)}\right) + (1 - y^{(i)}) \log\left(1 - a^{[2](i)}\right) \right) \tag{6}$$

Gradient descent

Recap: $\hat{y} = \sigma(w^T x + b)$, $\sigma(z) = \frac{1}{1+e^{-z}}$

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



Gradient Descent

Summary of gradient descent

$dz^{[2]} = a^{[2]} - y$	$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = dz^{[2]} a^{[1]^T}$	$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]^T}$
$db^{[2]} = dz^{[2]}$	$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$
$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(\mathrm{z}^{[1]})$	$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(\mathrm{Z}^{[1]})$
$dW^{[1]} = dz^{[1]} x^T$	$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$
$db^{[1]} = dz^{[1]}$	$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

Hidden Layers

- Training data does not specify the behavior of these layers
- Training data does not show the desired output of these layers
- Each hidden layer is vectorized value
- Dimensionality of these layers determine width of the model

Activation Function

- Sigmoid- 1/(1+e^-x), if output is binary choose sigmoid
- Hyperbolic Tangent – (e^z-e^-z)/(e^z+e^-z), wors better than sigmoid because the mean of its output is closer to zero, and so it centers the data better for the next layer.
- Rectified linear unit- Relu(z)=mac(0,z), superior to both Sigmoid and tanh because for both the gradient becomes very small when value of activation functionis very large or very small

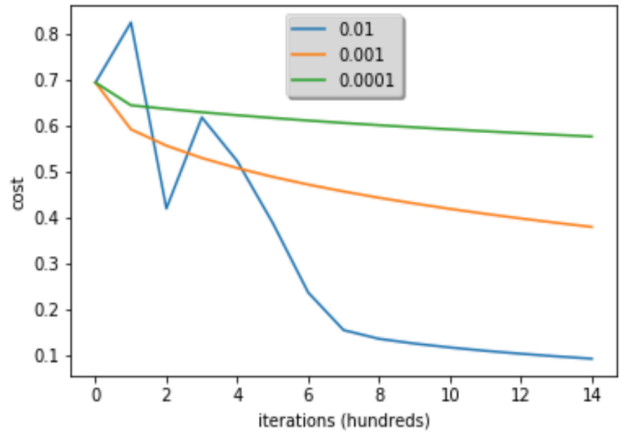
Initialization of Parameters

- Initialize weights to small random values
- if weights are initialized to zero, each neuron in the first hidden layer will perform the same computation. So even after multiple iterations of gradient descent each neuron in the layer will be computing the same thing as other neurons.
- Initialize biases to zeroes
- Learning rate should not be too large or too small
- If learning rate is too large it may overshoot the optimal value
- If learning rate is too small, there will be too many iterations required to converge to a value

```
-------------------------------------------------------

learning rate is: 0.0001
train accuracy: 68.42105263157895 %
test accuracy: 36.0 %

-------------------------------------------------------
```

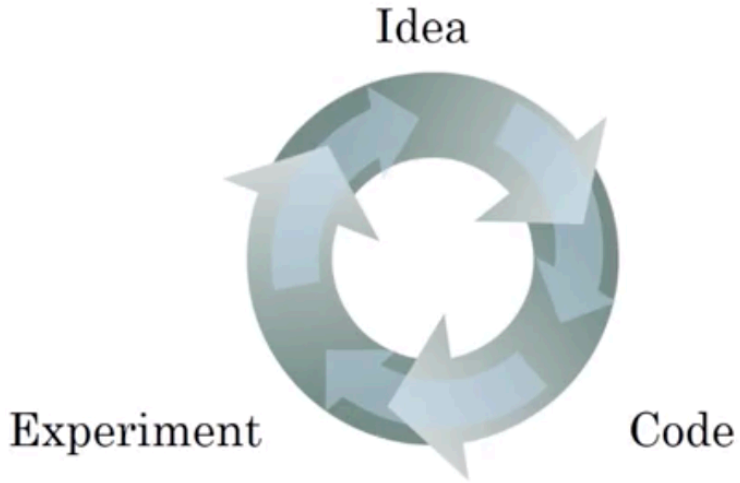


Cat Classification

- (num_px, num_px, 3) represents height, width, 3 channels for RGB
- reshape images of shape (num_px, num_px, 3) in a numpy-array of shape (num_px ∗∗ num_px ∗∗ 3, 1)
- divide every row of the dataset by 255
- Initialize the model's parameters
- Loop:
  - Calculate current loss (forward propagation)
  - Calculate current gradient (backward propagation)
  - Update parameters (gradient descent)

Hyperparameters

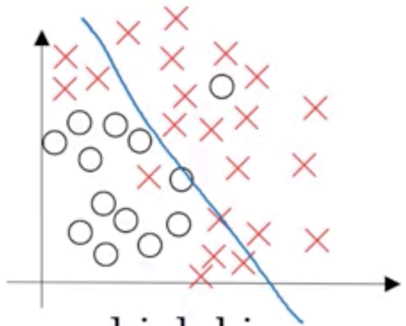
- Something which cannot be learned from data
- #layers
- #hidden units
- Learning units
- Activation functions
- Applied ML is a iterative process to understand best parameters to use

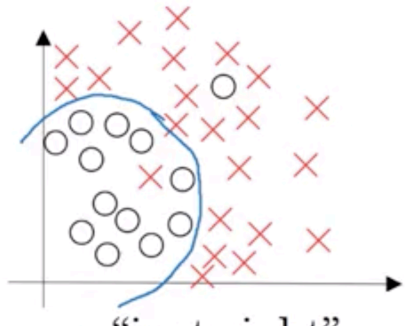


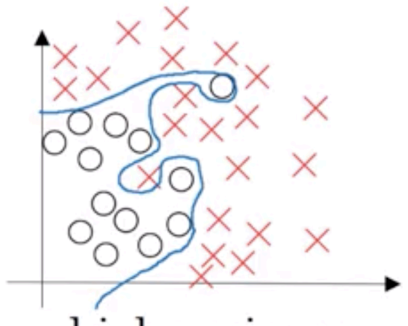
Mismatched train/test distribution

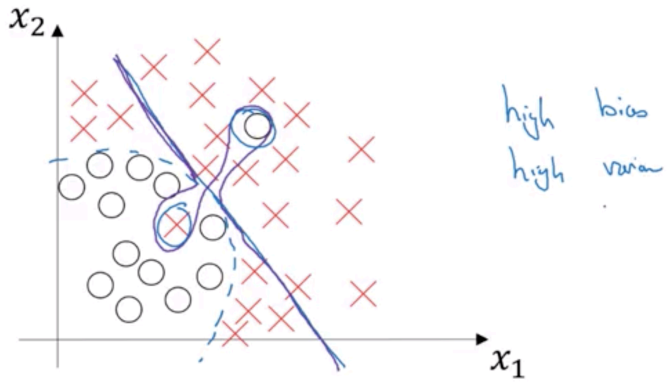
- Prediction works better when training and test set are from same distributions. For e.g.
- Training set: cat pictures from webpages (high resolution images)
- Test set: cat pictures from users in your app (low resolution images taken from phone)
- Not having a test set might be okay (only dev set)

Bias and Variance









- High bias- underfitting
- High variance - overfitting

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: ~0%

Workaround

- High bias(training data performance) - Bigger network, NN architecture search
- High Variance(dev set performance) – More data, Regularization, NN architecture search

Hyperparameter-Regularization

- Any modification to learning algorithm which reduces generalization error but not its training error
- E.g. adding restrictions on parameter values, add extra terms on objective function
- Adding extra constraints and penalties can be inspired by some prior knowledge
- Reduce variance significantly, not overly increasing the bias

Regularization

$$J(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta)$$

- Alpha is hyperparameter
- Larger value of alpha corresponds to larger regularization
- Penalize weights, leave bias unregularized
- Regularizing the bias parameters can introduce a significant amount of underfitting

L2 Parameter Regularization

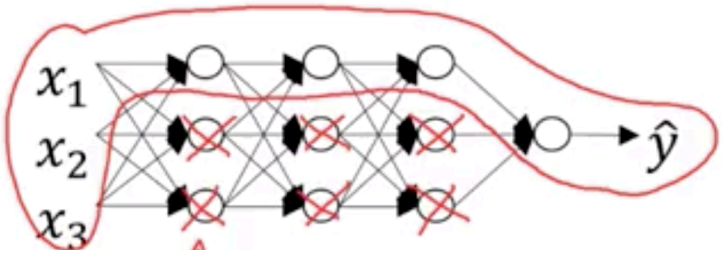
- Known as weight decay
- Drives the weight closer to the origin by adding a regularization term

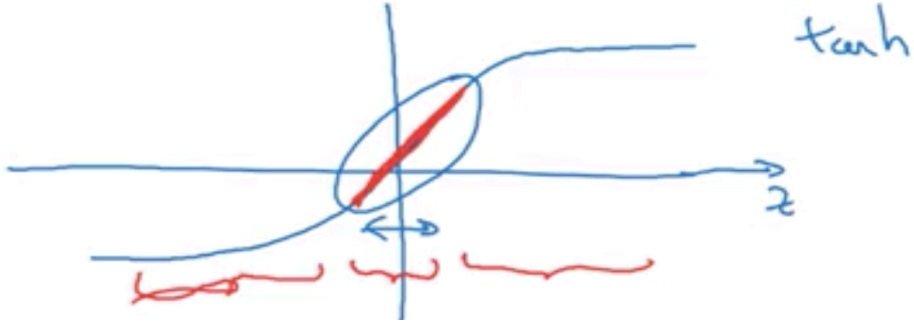
$$\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2$$

How Regularization prevents overfitting

- Setting lambda to be large, sets $w^{[l]}$ to be close to zero
- for some hidden units which makes network behave like a logistic classifier, hence making it very simple and reducing variance

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

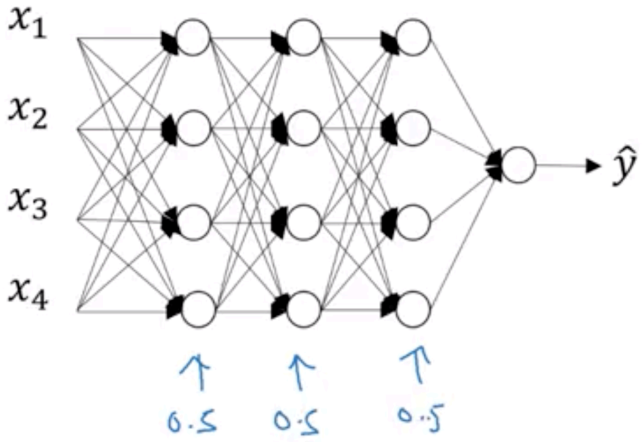


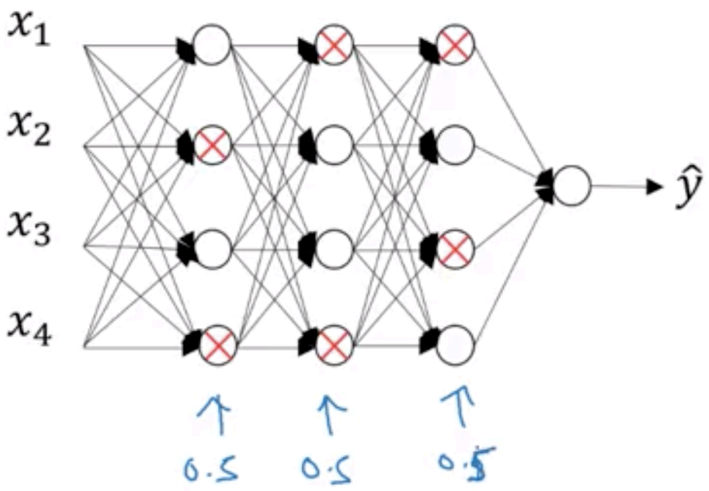


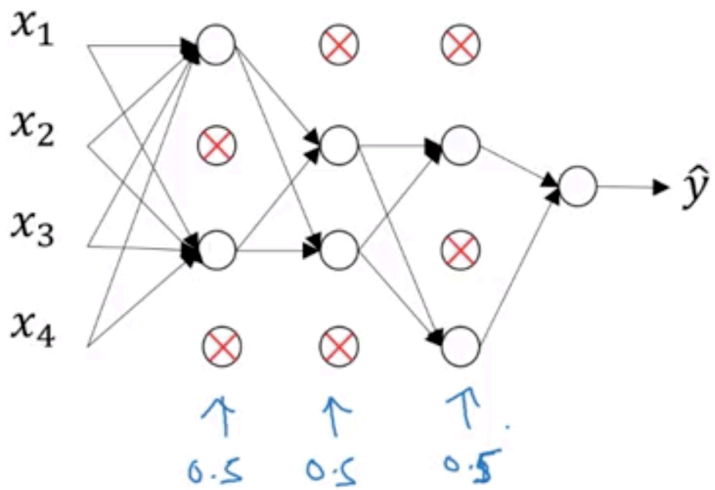
- w-small then z- small. So for tanh, if z is small, dz is almost linear, making the model simpler, hence reducing overfit

Dropout Regularization

- For each hidden layer, have 0.5 of keeping each node
- Eliminate those nodes and remove all ingoing and outgoing edges from that node
- Backpropagate on this network







Implementing Dropout (Inverted dropout)

- Illustrate with layer l=3 keep_prob=0.8
- d3= np.random.rand(a3.shape([0],a3.shape[1])< keep_prob
- d3 is Boolean array represented by 0 and 1
- a3= np.multiply(a3,d3)
- a3/=keep_prob
- 50 units originally→ 10 units removed (20% off)
- $Z^{[4]=} w^{[4]}. a^{[3]} + b^{[4]}$; $a^{[3]}$ reduced by 20%, by dividing by 0.8 it keeps the expected value of $a^{[3]}$ same
- Makes test time easier, less scaling problem
- Zero out different set of hidden units with every iteration

Regularization: Making Predictions at test time

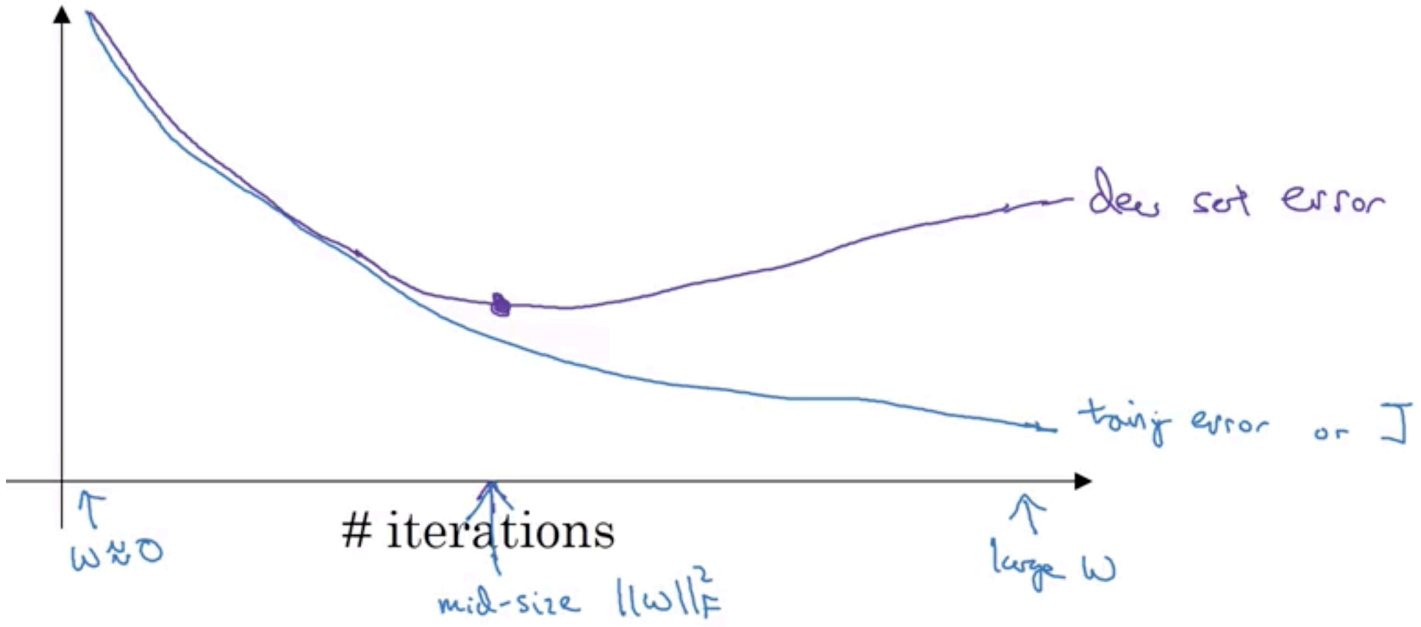
- $a^{[0]} = X$
- Use no drop out during test time
- This will add noise and output will be random which is not desired
- Keep less prob for layer with large number of hidden units
- Keep high prob for layers with less number of hidden units
- Highly used in computer vision because larger set of pixels
- Downside is cost function J is no longer defined because randomly dropping out some nodes

Regularization: Data Augmentation

- Data Augmentation: E.g. Flipping images to double size of training examples or randomly zoom in the image

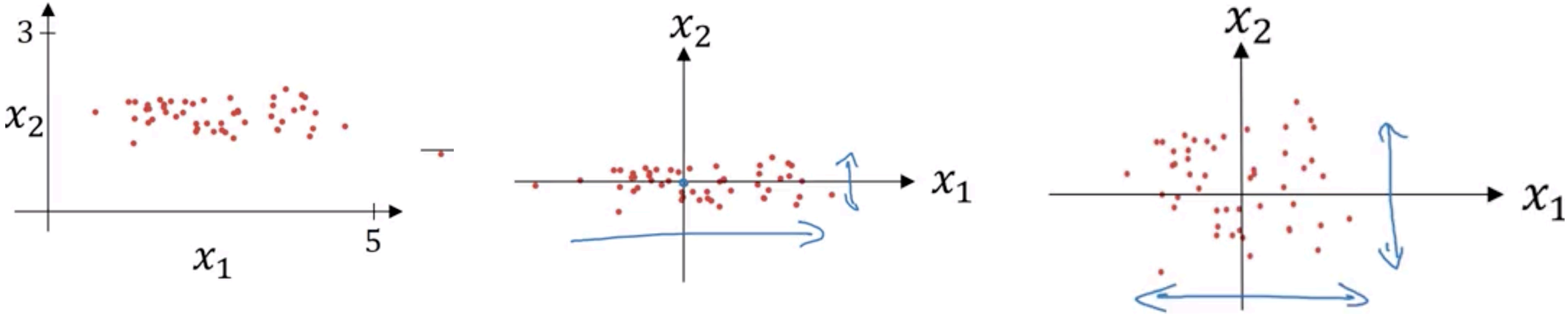
Regularization: Early Stopping

- Plot training set error vs #iterations
- Plot dev set error vs #iterations
- Stop neural network where dev set error increases



Normalizing Inputs

- Subtract mean: $\mu = \frac{1}{m}\sum_{i=1}^{m} x^i \ ; x = x - \mu$

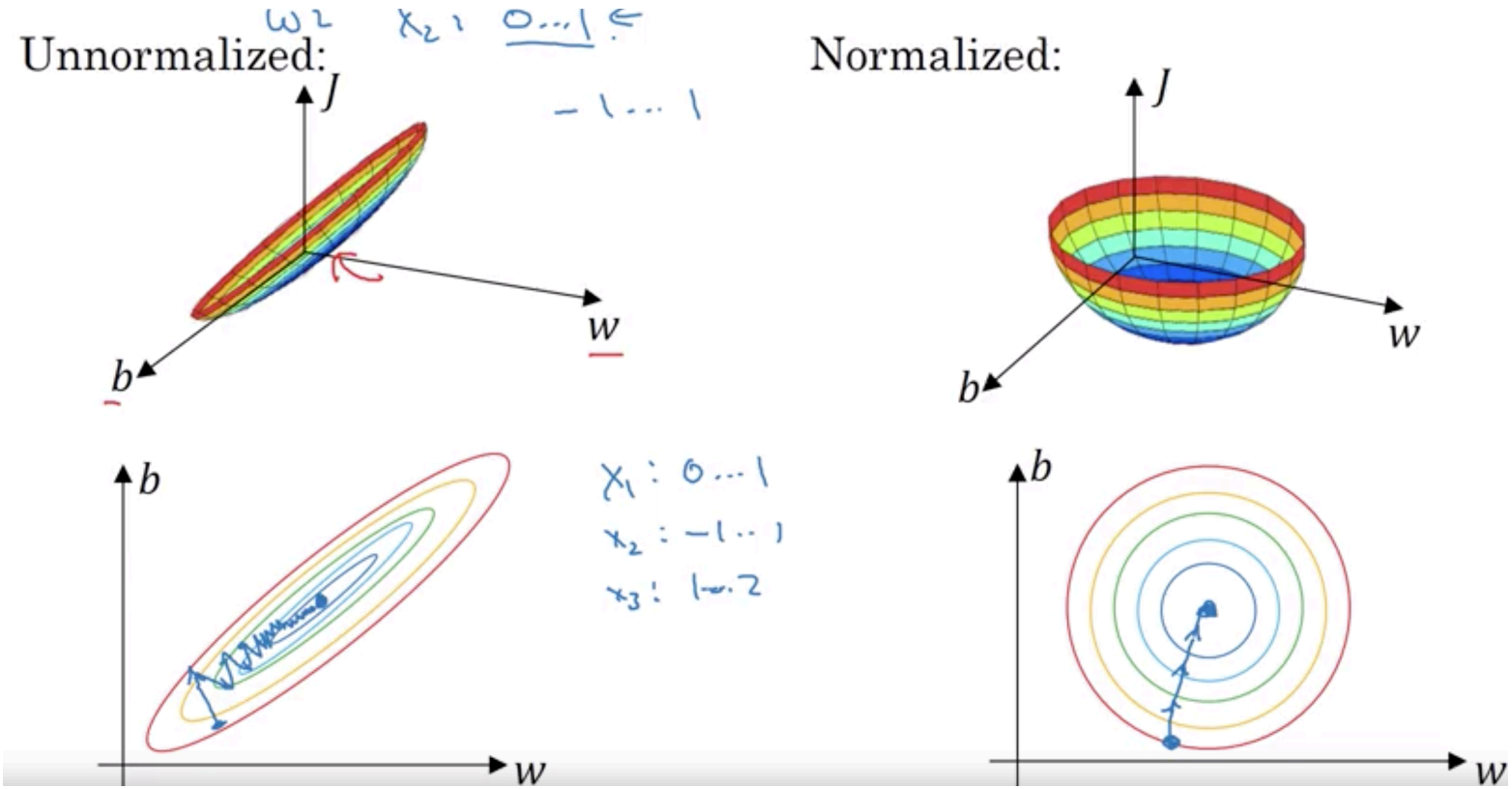


- Normalize mean: if variance of x_2 is much more than x_1

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^i \ ** 2 \ ; x /= \sigma^2, \ ** \text{ is Element wise multiplication}$$

Why normalized inputs

- If Unnormalized the cost function will vary more and look elongated
- Will have to use small learning rate if data is normalized



Plan for next week

- Complete optimization
- Complete CNN
- Complete Hyperparameter Tuning
- Read about CNN architectures

References

- Neural Networks and Deep Learning: https://www.coursera.org/learn/neural-networks-deep-learning/home/welcome
- Deep Learning book: http://www.deeplearningbook.org/
- Loss functions: https://isaacchanghau.github.io/post/loss_functions/
- Broadcasting: https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html
- Demystifying Deep CNN: http://scs.ryerson.ca/~aharley/neural-networks/
- Implementing a neural network from scratch in Python: http://www.wildml.com/2015/09/implementing-a-neural-network-from-scratch/
- Numpy: https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.shape.html
- Numpy.zeroes: https://docs.scipy.org/doc/numpy/reference/generated/numpy.zeros.html